Algorithms and Combinatorics of Repetitions in Strings

Toopsuwan, Chalita

Awarding institution:
King's College London

"Algorithms & Combinatorics of Repetitions in Strings"

Chalita Toopsuwan

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

Department of Informatics
School of Natural & Mathematical Science
King's College, London

September 2016

Acknowledgment

I would like to express my special appreciation and thanks to my advisor Prof. Maxime Crochemore, for the continuous support of my Ph.D study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D study. My second supervisor, Prof. Costas Iliopoulos, thank you for any kind of supporting and friendly relationship throughout my time at King's.

I am also grateful to my collaborators: Dr. Golnaz Badkobeh and Dr. Manal Mohamed. Thank you for encouraging our work and research until the ended of absolute papers.

Being a lonely aboard student in UK is not easy, thanks for the opportunity to meet another Thai colleague: Dr. Supaporn Chairungsee. She is not only collaborator but also one of my family.

I would like to express my sincere thank to Nicola Parsons, Nikki Nayler, Claudia Mazzoncini and many people in the Department of Informatics at Kings have offered support in various forms. There are too many people to mention by name, but I am grateful to them all. I would like to thank in particular Dr.Sanjay Modgil for being examiner of transferring my MPhil to PhD. and your so far kindly support and friendship.

A special thanks to my family. Words cannot express how grateful I am to my mother, and father for all of the sacrifices that you have made on my behalf. My grand parents, they are always my inspiration to achieve the degree. I would also like to thank all of my friends in UK who supported me through my far from home time, and incented me to strive towards my goal.

Abstract

Repetitions in strings constitute one of the most fundamental areas of string combinatorics with exactly essential applications to text algorithms, data compression, and also analysis of biological sequences. It is relevant to periodicities, regularities, and compression. The higher compression rate can be obtained from the repetitive behavior of strings, and reversely some compression techniques are at the core of fast algorithms for detecting repetitions.

Repetitions are highly periodic factors (or substrings) in strings, there are various type of repetitions such as repeat, repetition, squares, cubes, palindrome, maximal periodicitiie which is also called runs. The aim of this thesis is concentrated on the repetitions in strings in algorithmic and combinatorics approaches as they are very intricate and plenty of interesting works remain as open problems.

The critical study of this thesis firstly approach to the maximal periodicities or runs. It presents in Algorithmics of repetitions, local periods and critical factorization. An algorithm is designed in order to compute all runs for a string drawn from an infinite alphabet. On a string of length n, the algorithm runs optimally in time $O(n \log n)$ while there is a linear number of runs. The key model of computation is the comparison of letters which is done with the equality operator only. Under the same proposition, another time-optimal algorithm is created. This gives the same running time to compute local periods and all critical factorisations. The prefix table of input strings is applied as the main tool of those algorithms. In this study, we also design a simple algorithm based on the Dictionary of Basic Factors of the input string.

The notion of Gapped Palindrome and its Anti-exponent goes toward this research. A palindrome is a string $x = a_1 \cdots a_n$ which is equal to its reversal $\widetilde{x} = a_n \cdots a_1$. The definition of a gapped palindromes is given by a string of the form $uv\widetilde{u}$, where u, v are strings, $|v| \geq 2$, and $\widetilde{u}$ is the reversal of u. Replicating the standard notion of string exponent, we together define the anti-exponent of a gapped palindrome $uv\widetilde{u}$ as the quotient of $|uv\widetilde{u}|$ by $|uv|$. In this work, an algorithm is described to compute the maximal anti-exponent of

gapped palindromes occurring in an ordinary palindrome-free string. To get an efficient computation of maximal anti-exponent of factors in a palindrome-free string, we apply techniques based on the suffix automaton and the reversed Lempel-Ziv factorisation. The complexity analyse shows that algorithm runs in linear-time on a fixed-size alphabet.

Repeats are also of main concern in the domains of text compression and of pattern matching so lastly the study of repeat and its exponents are discussed in this thesis. Here we create linear-time algorithm to compute maximal exponent of repeats occurring in an overlapping-free string. Two main tools for the algorithm are a factorisation of the string and the Suffix Automaton of some factors. Eventually, we obtain the graceful result as the direct consequence from this research. There is the linearity of the number of occurrences of repeats whose exponent is maximal in an overlap-free string.

Among all of the previous researches and our further viewpoints in this thesis, acquiring knowledge on repetitions in string remains interesting open questions to continue.

Thesis Declaration

I hereby certify that this doctoral thesis represent my own work which has been done after registration for the Doctor of Philosophy degree in Computer Science of Department of Informatics, King's College London since April 2011. The next page shows the list of articles that were published during my period of research time. Certain material and concepts from these publications are necessarily presented within the body of this work.

Signed : Chalita Toopsuwan

Student ID : 0983336

Date : 1 September 2016

Publications

During the work on this thesis, the following related articles have been published by the author.

- Golnaz Badkobeh, Maxime Crochemore, Manal Mohamed, and Chalita Toopsuwan. Maximal anti-exponent of gapped palindromes. *Theoretical Computer Science*, 2016. In press.

- Golnaz Badkobeh, Maxime Crochemore, and Chalita Toopsuwan. Computing the maximal-exponent repeats of an overlap-free string in linear time. In L. Calderon-Benavides, C. Gonzlez-Caro, E. Chvez and N. Ziviani, editors, *Symposium on String Processing and Information Retrieval*, number 7608 in LNCS, pages 61-72. Springer, 2012. **This article was awarded the Best Paper of SPIRE 2012**

- Golnaz Badkobeh, Maxime Crochemore, and Chalita Toopsuwan. Maximal anti-exponent of gapped palindromes. In *Fourth International Conference on Digital Information and Communication Technology and its Applicationsm DICTAP 2014, Bangkok, Thailand, May 6-8, 2014*, pages 205-210. IEEE, 2014.

- Maxime Crochemore, Tomasz Kociumaka, Wojciech Rytter, Chalita Toopsuwan, Wojciech Tyczyński, and Tomasz Waleń. *Algorithmics of repetitions, local periods and critical factorisation revisited*, pages 53-60. Czech Technical University, Prague, 2012.

Contents

List of Figures

1

Introduction

Repetitions in strings constitute one of the most fundamental areas of string combinatorics with exactly essential applications to text algorithms, data compression, and also analysis of biological sequences. It is relevant to periodicities, regularities, and compression. The higher compression rate can be obtained from the repetitive behavior of strings, and reversely some compression techniques are at the core of fast algorithms for detecting repetitions.

Repetitions are highly periodic factors (or substrings) in strings, there are various type of repetitions such as repeats, repetitions, squares, cubes, palindrome, and maximal repetitions which is also called runs.

In my research area, I am concentrated on the repetitions in strings in algorithmic and combinatorics approaches as they are very intricate and plenty of open problems are remained.

The study of repetitions together with period has already been studied by Axel Thue [55] in 1906, regarded as discovery of stringology. His concentration was searching long sequences with few repetitions, however in the algorithmic framework, one of the difficulties was finding all repetitions fast. The strategy of construction of linear-time algorithms to encode all repetitions in strings has been assigned as the algorithmic side of the problem.

As there are several kinds of repetitions in strings, a number of previous

works from many researchers are now presented.

Maximal periodicities, called runs, capture consecutive repetitions in strings. In 1981 the problem of runs was firstly studied by Maxime Crochemore [10], his work introduced maximal (non-extendable) integer powers and an $O(n \log n)$ algorithm for finding them all. Applying the Fibonacci strings, the optimal bound was additional presented.

The next research movement was considering occurrences of fractional repetitions of right-maximal repetitions (non-extendable to the right repetitions). It was done by Apostolico and Preparata in 1983 [1]. Then the occurrences of fractional repetitions of maximal repetitions was studied by Main [45]. The later presented a linear-time algorithm for finding all leftmost occurrences of runs.

Then Iliopoulos et al. [34] showed that for Fibonacci strings, the number of maximal repetitions is linear. Although Iliopoulos and his team applied this technique to a particular class of strings, it is important since the Fibonacci strings were known to contain many repetitions.

The vital element of any algorithm computing all repetitions in strings of length n in $O(n)$ time is the fact that the number of maximal repetitions (runs) is linear. Therefore, the most important part of the analysis of the running time of such algorithms is counting the number of runs. In a two-decade time, there were plenty of consequences effort in the stringology community to find such algorithms.

At the end of the second millennium (see [41, 38]), the invention was approached by Kolpakov and Kucherov, where it was finally proved that encoding all occurrences of repetitions into runs was the right way to obtain a linear-sized output.

Kolpakov and Kucherov provided two results, the first one is applying previous techniques of Crochemore [12], Main and Lorentz [45], and Main [44] to construct an algorithm that computes all maximal repetitions (or runs) in time proportional to the size of the output. Moreover they proved that the maximum number of runs in a string of length n, $RUNS(n)$, is linear, i.e. $RUNS(n) \leq cn$, for a constant c. This result was the crucial contribution however, they could

n	5	6	7	8	9	10	11	12	13	14
Max. no. of runs	2	3	4	5	5	6	7	8	8	10

15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
10	11	12	13	14	15	15	16	17	18	19	20	21	22	23	24	25

Figure 1.1: Maximum number of runs in binary strings of length $n, 5 < n < 31$

not provide any bound on the constant c. Nevertheless, the following conjecture could be stated for binary alphabets, based on the numerical evidence, see Figure1.1.

Conjecture 1 (*The Runs Conjecture*) For any $n \geq 1$, $RUNS(n) \leq n$.

Another breakthrough came recently in 2006, when Rytter [51] was able to present a bound for such constant c by giving $c \leq 5$. After that, Puglisi et al. [48] developed Rytter's analysis, they reduced the value of c to 3.48. Rytter ended up the year with this kind of research by producing his own version of the improved analysis with a constant factor of 3.44, see [52].

A more powerful conjecture is proposed by Franek et al. [30] where a family of strings is given with the upper limit of the number of runs equal to $\frac{3}{2\phi}n = 0.927\dot{n}$, where ϕ is the golden ratio, thus for proving $c \geq 0.927$. They presented a conjecture stating that this kind of bound is optimal. However the conjecture is false since the lower bound has gone from that value to 0.944565 in the work of W. Matsubara et al. [47]. They initially showed a concrete string τ of length 184973, which contains 174697 runs in it. It immediately disproves the conjecture, since $174697/184973 \approx 0.944445$ is already higher than the previous bound 0.927 of Franek. Then they proved that the string τ^k, which is the string obtained by concatenating k copies of τ, contains $174719k - 21$ runs for any $k \geq 2$. Since $|\tau^k| = 184973k$, it yielded the new lower bound $174719/184973 \approx 0.944565$ as $k \to \infty$.

Recently J. Simpson [54] produced ternary words called Modified Padovan words, then applied a morphism to these to produce run-rich binary words.

With the interesting properties of Modified Padovan words, they improved the lower bound to be 0.944575712.

After that, Maxime and Ilie improved Franek's result by showing a bound for the constant c in the conjecture; $c \leq 1.6$, hence the number of runs in a string of length n is at most $1.6n$, see [14].

In their study coordinating with Liviu, they applied a combination of theory and computer affirmation, and presented a closer bound. They eventually found that $RUNS(n) \leq 1.029n$, see [15].

In summary, we now obtain that there are at most $1.029n$ and at least $0.944575712n$ maximal repetitions in a string of length n. In the mean time, the conjecture that n is the exact bound is still unsolved. There are a number of research work coming closer to solve the conjecture. So more improving studies have been kept going.

The local periods of a string, like its runs, capture the repetitions in the string. They have to do with its critical factorisations or critical positions, where the local period equals the global smallest period of the string. The computation of local periods is known to be done in linear time using a data structure that depends on an alphabet ordering [26]. There are other algorithms for computing such repetitions or local periods of words in linear time, however all these algorithms require again an ordered alphabet. One example is presented in [16]), where instead of suffix trees, authors utilise suffix arrays. In term of local periods, they create an algorithm as powerful as in [26] but in a simpler way applying the solution of the Manhattan Skyline Problem.

Contribution : In Chapter 3, we provide an algorithm for computing all runs in a length-n string y in time $O(n \log n)$ in the model of computation where the comparison of letters is done with the equality operator only. This gives the same running time to compute local periods and all critical factorisations of the string y on an unordered alphabet. This $O(n \log n)$-time local period computation is optimal since it implies square detection. The main tool we used in algorithms is the prefix table of the string. In this chapter, we also design a simple algorithm based on the Dictionary of Basic Factors of the input string.

A palindrome is a string that reads the same backward and forward, that is a string x which $x = \widetilde{x}$, where $\widetilde{x}$ called *reversal* of x.

The purpose of this thesis is to study the *gapped palindrome*, which is a word of the form uvu^T for some word u, v with $|v| \geq 2$ and u^T denotes the reversal of u. The word u and u^T are named as left arm and right arm of the palindrome respectively, and v is also known as a *gap* or *spacer*.

The maximal gapped palindrome verifies more condition of their arms which cannot be extended outward or inward preserving the palindrome structure. Gusfield [33] presented a algorithm for computing gapped palindrome. Even if the algorithm runs in linear time, it applied to only fixed spacer length of the palindromes.

In 2008 Kolpakov and Kucherov extended this kind of work by proposing algorithms for computing two classes of gapped palindromes. The first class requires the condition of $|v| \leq |u|$ and is called long-arm palindrome. The other class is called length-constrained palindromes defined by bounded spacer length, and a lower bound on the arm length, i.e. $MinGap \leq |v| \leq MaxGap$ and $MinLen \leq |u|$ for some constants $MinGap, MaxGap$, and $MinLen$. Both classes verify the maximal condition of gapped palindromes. Their algorithms applied to both classes and run in time $O(n + S)$, where n is the length of the input word and S is the number of output palindromes [39]. However they have not counted the number of such gapped palindromes occurring in a string.

In the other way, the study of finding the longest previous reverse factor occurring at each position of a string is presented in [8]. This is a principal notion used for the optimal detection of various types of palindromes in a string. Their work is described in two algorithms for computing the LPrF table of a string, the first one is obtained from its Suffix Tree while the second from its Suffix Automaton. Both algorithms run in linear time on a fixed size alphabet. Typical applications of these algorithms are for RNA secondary structure prediction and for text compression.

The LPrF table stores each index i the maximal length of factors starting at position i on the given string and with their reverse occurring at a smaller position. Moreover, the algorithm for LPrF table can be computed in linear

time on a fixed size alphabet.

Contribution: In Chapter 4, the standard notion of string exponent is replicated. The *anti-exponent* of a gapped palindrome $uv\widetilde{u}$ is defined as the quotient of $|uv\widetilde{u}|$ over $|uv|$. We apply techniques based on the suffix automaton and on the reversed Lempel-Ziv factorisation to an input string x containing no ordinary palindrome, for which if factor $uv\widetilde{u}$ appears in y, v is not a palindrome, and design an algorithm to compute the maximal anti-exponent over all gapped palindromes of the given string. Our algorithm runs in linear-time on a fixed-size alphabet in contrast to a naive cubic time solution.

Repeats is another interesting kind of repetitions. *Repeats* considered in this thesis is the strings of exponent at most 2. They refer to strings of the form uvu where u is its longest *border* (both a prefix and a suffix). The study of repeats in a string has to do with long-distance interactions between separated occurrences of the same segment (the u part) in the string. Although occurrences may be far away from each others, they may interact when the string is folded as it is the case for genomic sequences. A very close problem is that of computing maximal pairs (positions of the two occurrences of u) as described by Gusfield [33], which apply a suffix tree to report all maximal pairs in a string in time $O(n+z)$ and space $O(n)$, where n is the length of the string and z is the number of output pairs. Nevertheless, Gusfield's algorithm determines no constraints on the gaps of the output pairs. That means their occurrences can be either overlapping or far apart in the string.

A later improved work by Brodal et al. [6] devoted to gap restrictions in the work of finding maximal pairs with a bounded gap. In this research, various restrictions on the gap were given. They applied the fundamental data structure of suffix tree in all work and used stronger version of the "smaller-half tricks" to find all maximal pairs with a gap in an both upper and lower bounded interval that they can find them in time $O(n \log n + z)$ where n is the string's length and z is the number of output pairs. They also put less restriction on the gap by searching for maximal pairs with only a lower bound set. Applying a heap-ordered tree in this kind of single bound gap, the running time decreases to $O(n + z)$.

Contribution : Chapter 5 presents an algorithm to compute repeats obtaining the maximal exponent value in an overlap-free string. In this work, we use two main tools: a factorisation of the string and the Suffix Automaton of some factors. Our algorithm runs in linear-time on a fixed-size alphabet, while a naive solution of the question would run in cubic time. The solution for non overlap-free strings derives from algorithms to compute all maximal repetitions, also called runs, occurring in the string.

As an additional result of this study, we show there is a linear number of maximal-exponent repeats in an overlap-free string. The algorithm can locate all of them in linear time.

A square is a repetition with exponent $r = 2$, that is a string of the form ww, where w is nonempty. In the investigation of occurrences of squares, their number is $\Theta(n \log n)$ in a string of length n and there is a linear time algorithm to compute all of them as the potential size of the output, see [10].

Different from the number of squares' occurrences, Fraenkel and Simpson [28] presented that only $O(n)$ distinct squares can appear in a string of length n and refined the upper bound by $2n$. Meanwhile, experiment motivates the belief in the fact that this maximal number is strictly less than n once again.

For this kind of repetitions, the problem of three overlapping occurring in a string was initially studied by Crochemore and Rytter [21] in the "Three Squares Lemma" presenting that three squares could exist at the same position in a string only if the longest of the three was at least the sum of the lengths of the other two.

In "A new periodicity lemma"; the paper of Fan et al. [27] , the lemma was further generalized, they presented a periodicity lemma showing that the occurrence of two squares at the same position in the string x, together with the occurrence of a neighboring third square to the right position, is possible only in specific situations. At that time, the problem **P** is considered:

(**P**) "Suppose that a string x has prefixes u^2 and v^2 with $\frac{3|u|}{2} < |v| < 2|u|$, and suppose moreover that a third square w^2 occurs at position $k + 1$ of x, where $|v| - |u| < |w| < |v|, |w| \neq |u|$, and $0 \leq k < |v| - |u|$," what can be detailed for the periodicity of the string x?

The problem **P** was studied by the assumption that $x = v^2$ has prefix u^2, if $\frac{3|u|}{2} < |v| < 2|u|$, then a breakdown of the string x is arisen by; $x = u_1u_2u_1u_1u_2u_1u_2u_1u_1u_2$ with size of u_1, u_2 are positive that $|u_1| = 2|u| - |v|$ and $|u_2| = 2|v| - 3|u|$.

Then all possible occurrences of u in x were classified into 14 subcases that determined by the value of parameters $|u|, |v|, |w|$, and k as shown in the Figure 1.2.

Thereafter Kopylova and Smyth [42] extended those results to a case not previously considered, then go on to make the results of the new periodicity lemma more precise under weaker assumptions. They described experiments conducted on strings that satisfy the three squares condition, then used the results of these experiments to formulate conjectures about the nature of x and its alphabet in each of the 14 subcases as detailed in Figure 1.3. And finally they proved the correctness of seven of these conjectures.

In the paper [29], Smyth continued this kind of work with others researchers by providing a unifying framework for his previous results and showing that in 12 of 14 subcases that arise, the postulated occurrence of three neighbouring squares forces a breakdown into highly periodic substrings. They moreover have proved the subcase 4 and clarified the results for subcases 11-14. At that time, the periodicity of x was resolved for all subcases except case number 3 and 7.

Smyth has recently presented a new result to this research area by expressing how to characterize the general case of overlapping squares without the restriction of two squares beginning at the same position.

He has a conclusion in the present work that the analysis given in the result of previous papers [27, 53, 42, 29], deals with 12 of 14 subcases that arise: two remain to be considered, but it seems clear that such behaviour is impossible that is, the assumption that three neighbouring squares of well-defined size exist within these well-defined bounds leads to the conclusion that the string locally breaks down into repetitions of small period.

Even though the study of squares is not significantly presented in the thesis, this kind of repetition is one of the interesting repetition and remains a number of open problems to study.

S	k	k+w	k+2w	Conditions
1	$0 \leq k \leq u_1$	$k+w \leq u$	$k+2w \leq u+u_1$	$k \geq u_2$
2	$0 \leq k \leq u_1$	$k+w \leq u$	$k+2w \leq u+u_1$	$k < u_2$
3	$0 \leq k \leq u_1$	$k+w \leq u$	$k+2w > u+u_1$	-
4	$0 \leq k \leq u_1$	$u < k+w \leq u+u_1$	-	-
5	$0 \leq k \leq u_1$	$u+u_1 < k+w \leq v$	-	-
6	$0 \leq k \leq u_1$	$v < k+w < 2u$	-	-
7	$u_1 < k < u_1+u_2$	$k+w \leq u+u_1$	$k+2w \leq 2u$	-
8	$u_1 < k < u_1+u_2$	$k+w \leq u+u_1$	$k+2w > 2u$	-
9	$u_1 < k < u_1+u_2$	$u+u_1 < k+w \leq v$	-	$w < u$
10	$u_1 < k < u_1+u_2$	$k+w \leq v$	$k+2w \leq u+v$	$w > u$
11	$u_1 < k < u_1+u_2$	$k+w \leq v$	$u+v < k+2w \leq 2v-u_2$	-
12	$u_1 < k < u_1+u_2$	$k+w \leq v$	$2v-u_2 < k+2w$	-
13	$u_1 < k < u_1+u_2$	$v < k+w \leq 2u$	-	-
14	$u_1 < k < u_1+u_2$	$2u < k+w < 2u+u_2-1$	-	-

Figure 1.2: The 14 subcases for three neighbouring squares u, v, and w while $v-u < w < v, w \neq u$, and $0 \leq k < v-u$.

Subcases S	Conditions	Breakdown of x/v^2
1,2,5,6,8-10	$(\forall x, \sigma = d)$	$x = d^{x/d}$
3,4,7	$\sigma = d$	$x = d^{x/d}$
	$\sigma > d$	$x = s^{\alpha} s[1..u_1 \bmod s] s^{\gamma} s[1..u_1 \bmod s] s^{\varepsilon}$
11-14	$\sigma = d$	$x = d^{x/d}$
	$\sigma > d$	cannot determine

Figure 1.3: Overview of the conjectures that α is the largest alphabet size compatible with the particular parameters u, v, k, and w, when $d = gcd(u_1, u_2, w) = gcd(u, v, w)$ and $s = gcd(u-w, v-u)$ for $\alpha = \lfloor u/s \rfloor, \gamma = \lfloor v/s \rfloor$, and $\varepsilon = (v-u)/s$.

2

Theoretical background

According to the strings' history in the previous chapter, repetitions are highly periodic factors in strings and there are various types of them. Repetitions of our interest are explicitly determined in this chapter. Besides those things, basic definitions and background knowledge of string theory that relate to our research are embraced.

1 Strings

A ***string*** (or ***word***) is a finite sequence of letters (or symbols) drawn from a finite (or infinite) set A called the ***alphabet***. The alphabet size is $a = |A|$, we denote A^* to be the set of all finite strings from the set A. Let x be a string, $|x|$ denotes the ***length*** of x. If i and j are two integers for which $0 \leq i, j < |x| - 1$, $x[i]$ stands for the letter at position i of x, a ***substring*** (or factor) of x is of the form $x[i]x[i+1] \dots x[j]$. The substring can be denoted by $x[i\,..\,j]$ and it is the empty if $j < i$. If $i \neq 0$ and $j \neq |x| - 1$, the factor is a ***proper substring*** of x.

The factor is also called a ***prefix*** of x if $i = 0$ and a ***suffix*** of x if $j = |x| - 1$. A prefix $x[0\,..\,j]$ is a ***proper prefix*** of x if $j \neq |x| - 1$. Similarly, a suffix $x[i\,..\,|x| - 1]$ with $i \neq 0$ is a ***proper suffix*** of x.

The string x has ***period*** p, $0 < p \leq |x|$, if $x[i] = x[i+p]$ whenever both sides of the equality are defined. The period of x, period(x), is its smallest period. The string x is said to be ***periodic*** if period(u) $\leq |u|/2$.

The ***exponent*** of x is the ratio between its length and period, that is $\exp(x) = |x|/\text{period}(x)$. Then, x can be written as u^e where u is its prefix of length period(x), and is called an e-power. The string x is called ***primitive*** if it is not an e-power for any integer $e \geq 2$. In other words x is primitive if none of its periods is a divisor of its length.

Example:

- $\exp(\texttt{mom}) = 3/2 = 1.5$
- $\exp(\texttt{abcabcab}) = 8/3 = 2.67$
- $\exp(\texttt{asgoodas}) = 8/6 = 1.33$

2 Repeats

A ***repetition*** in a string x is an interval $[i\,..\,j]$ for some integers $0 \leq i, j < |x|-1$ for which the associated factor $x[i\,..\,j]$ is periodic.

If $x[i\,..\,j] = u$, the repetition can be written in the form u^r where r is considered as the exponent of the repetition. Note that the exponent of the repetition is usually at least 2.

Example:

- $\texttt{abaab abaab abaab ab} = (\texttt{abaab})^{17/5}$
- $\texttt{alfalfa} = (\texttt{alf})^{7/3}$
- $\texttt{entente} = (\texttt{ent})^{7/3}$

One of the most analysed type of repetition is repetitions with even integer exponent. It is called as ***square***, that is a string of the form ww, where w is nonempty.

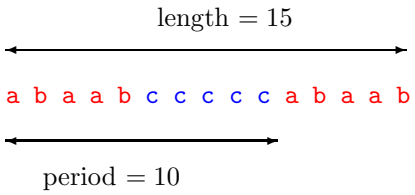
Example:

- `abcabc` $= (\texttt{abc})^2$
- `loglog` $= (\texttt{log})^2$

A ***repeating substring*** in a string x is a nonempty proper substring u of x whose occurrences appear more than once in x. For example, if $x = efeefefe$ is a string, then $u = efe$ is a repeating substring in x. Hence their occurrences may be nonadjacent or overlapping.

A ***repeat*** in x is a tuple $(u; i_1, i_2, .., i_r)$, where u is the repeating substring occurring at position $i_1, i_2, .., i_r$ in x with $0 \leq i_1 < i_2 < .. < i_r \leq |x| - 1$. According to the structure of repeat, its exponent is greater than 1 but not over 2.

Example:



- The above repeat has exponent $= \frac{\text{length}}{\text{period}} = \frac{15}{10} = 1.5$ and can be written as `abaab ccccc abaab` $= (\texttt{abaabccccc})^{15/10}$
- `all in all` $= (\texttt{all in }\ )^{10/7}$
- `restore` $= (\texttt{resto})^{7/5}$

3 Runs

A ***run (or maximal periodicity or maximal repetition)*** in a string x is a maximal (non-extendable) occurrence of a repetition of exponent at least two. That means for some position i and j in the string x, $[i \ldots j]$ is a run if

1. $x[i \ldots j]$ has period p,

2. $j - i + 1 \geq 2p$,

3. $x[i-1] \neq x[i+p-1]$ (if $x[i-1]$ is defined), $x[j+1] \neq x[j-p+1]$ (if $x[j+1]$ is defined) and

4. $x[i \ldots i+p-1]$ is *primitive*, that is, it is not a proper integer power two or any larger of another string.

We say for such a case that there is a run with period p *at* $i+p$ *in* x or *run* p *at* $i+p$ *in* x. Moreover, position $i+p$ is known as the *center* of the run.

Example:

For the string $x =$ `abaababaabaab`, the following are examples of runs in x;

- run $[3 \ldots 7]$ with period 2 and exponent $5/2$, that is, $x[3 \ldots 7] =$ `ababa` $= (ab)^{2.5}$
- run $[0 \ldots 10]$ with period 5 and exponent $11/5$, that is the prefix `abaababaaba` $= (abaab)^{2.2}$
- run $[5 \ldots 12]$ with period 3 and exponent $8/3$, that is the suffix `abaabaab` $^{2.67}$

while $x[3 \ldots 6] =$ `abab` is not.

4 Palindromes

A ***palindrome*** is a string that reads the same backward and forward that is a string x which $x = \widetilde{x}$, where $\widetilde{x}$ called reversal of x.

Example:

- $x = abba = \widetilde{x}$
- $x = abaaba = \widetilde{x}$

A ***gapped palindrome*** is a string of the form $uv\widetilde{u}$ for some substring u, v with $|v| \geq 2$. u and $\tilde{u}$ are said to be the **anti-borders** of the gapped palindrome if and only if u is the longest prefix of the gapped palindrome for which $\widetilde{u}$ is

a suffix of the gapped palindrome. The idea of string exponent is also defined to a gapped palindrome. It is called as **anti-exponent** and given by the ratio $|uvu|/|uv|$.

5 f-factorisation

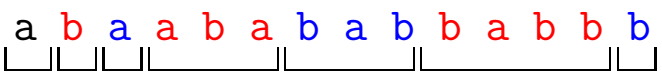
The core algorithms presented in this research rely on the f-factorisation of strings, a type of ***Lempel-Ziv 77 factorisation (LZ 77-factorisation)***, see [23].

For a given string y, the LZ 77-factorisation of y is a sequence of non-empty strings, z_1, z_2, ..., z_k for some positive integer k, called *phrases*, and satisfy the following properties:

- $y = z_1 z_2 \cdots z_k$ where,
- z_i is the longest prefix of $z_i z_{i+1} \cdots z_k$ occurring in $z_1 z_2 \cdots z_{i-1} z_i$, and
- if this longest prefix is empty, z_i is the first letter of $z_i z_{i+1} \cdots z_k$.

The LZ-factorisation of the string y can be computed in $O(|y|)$ times and requires also $O(|y|)$ spaces on a fixed-size alphabet using any suffix data structure, see [23].

The following figure shows the LZ 77-factorisation of string *abaababababbabbb* which is determined by the sequence z_1, z_2, ..., z_7 when $z_1 = a, z_2 = b, z_3 = a, z_4 = aba, z_5 = bab, z_6 = babb$, and $z_7 = b$.



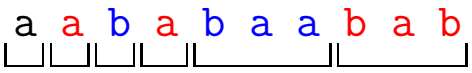
Lempel-Ziv 77 factorisation can also be defined in reverse version and called ***reversed LZ-factorisation***. The reversed LZ-factorisation of a string y (see [39]), is a sequence of non-empty strings, z_1, z_2, ..., z_k for some positive integer k, satisfy the following properties:

1. $y = z_1 z_2 \cdots z_k$ where,

2. z_i is the longest prefix of $z_i z_{i+1} \cdots z_k$ occurring in the reverse of $z_1 z_2 \cdots z_{i-1} z_i$, and

3. when that prefix is empty, z_i is the first letter a of $z_i z_{i+1} \cdots z_k$ (a does not occur previously in $z_1 z_2 \cdots z_{i-1}$.)

For example, the reverse LZ-factorisation of string *aababaabab* is the sequence $z_1 = a, z_2 = a, z_3 = b, z_4 = a, z_5 = baa$, and $z_6 = bab$ as shown below.



The computation of the reversed factorisation of a given string y can be done in $O(n)$ time and space by exploiting the suffix array and the LCP array, see [20]) for more information.

6 Suffix Automaton

An efficient data structure for representing a full index of a set of strings which is called automaton.

An ***automaton*** M on an alphabet set A is composed of a finite set S, of a state q_0, of a set $T \subseteq S$ and of a set $F \subseteq S \times A \times S$, called ***states, initial state, terminal states*** and ***arcs*** - or transitions respectively. The automaton M can be denoted by quadruplet:

(S, q_0, T, F). An arc $(p, a, q) \subseteq F \subseteq S \times A \times S$ stands for the transition that leaves the state p and enter the state q. State p is known as the ***source***, letter a as ***label*** and state q as ***target*** of the arc.

A type of automaton that we apply in this research is the suffix automaton.

The ***suffix automaton*** is a minimal deterministic automaton representing the set of all suffixes of a set of strings. We commonly use notation $\mathcal{S}(z)$ for the suffix automaton of z. Figure 2.1 shows the suffix automaton of the string `abcadbeca` for an example.

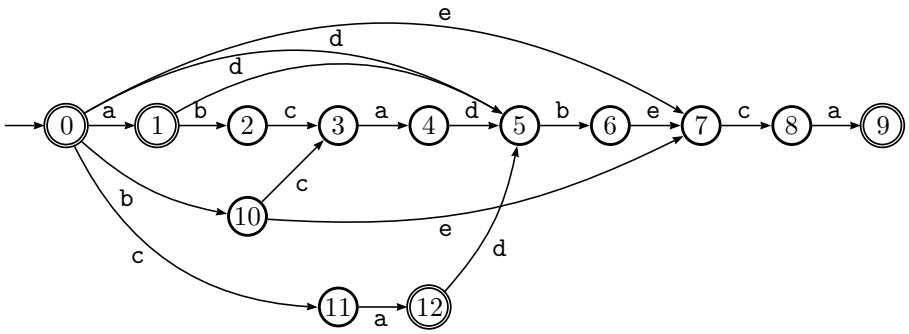


Figure 2.1: Suffix automaton of the string `abcadbeca`.

3

Algorithmic of repetitions, local periods and critical factorisation revisited

The study of repetitions and other structures of periodicity have been considered since the beginning of last century. A sample of methods and algorithm for detecting repetitions can be found in [33, Chapter 7] and [22, Chapter 8]. A run is a non-extensible occurrence of a repetition, that is, a maximal periodicity in the string. Main [44] considered leftmost periodicities, the concept of run is by Iliopoulos et al. [34], and the most significant algorithmic contribution was approached by Kolpakov and Kucherov [41]. It is known the number of runs in a string is linear with respect to the length of the string (see [41, 52, 15]). The exact bounds on the number of runs in a string is a fascinating open problem, especially interesting since the the number of repetitions can be $\Omega(n \log n)$.

The local periods of a string, like its runs, capture the repetitions in the string. They have to do with its critical factorisations or critical positions, where the local period equals the global smallest period of the string. The computation of local periods is known to be done in linear time using a data structure that depends on an alphabet ordering [26]. There are other algorithms for computing such repetitions or local periods of words in linear time, however all these algorithms require again an ordered alphabet. One example

is presented by Maxime et al., where instead of suffix trees, they utilised suffix arrays. In terms of local periods, they create an algorithm as powerful as that in [26] but in a simpler way applying the solution of the Manhattan Skyline Problem [16].

In this chapter we present an algorithm for computing all runs in a length-n string y in time $O(n \log n)$ in the model of computation where the comparison of letters is done with the equality operator only.

This gives the same running time to compute local periods and all critical factorisations of the string y on an unordered alphabet. This $O(n \log n)$-time local periods computation is optimal since it implies square detection. The main tool we used in algorithms is the Prefix table of the string.

The Karp-Miller-Rosenberg algorithm creates names for all factors with a power of two length. The created data structure is called the Dictionary of Basic Factors, a simple, powerful data-structure [19]. An additional result of this chapter is a simple algorithm based on the Dictionary of Basic Factors of the input string. The contribution paper for this chapter can be found in [17].

1 Runs in the concatenation of two words

In this section, divide and conquer is applied to compute runs and local periods of a string. Therefore, the main element of the algorithm has to do with the runs occurring when we concatenate two strings. We follow the approach initiated by Main and Lorentz [45] for detecting squares in strings (see also [12]).

Let $y = uv$ be the concatenation of two strings u and v. In this section, we show how to compute all runs of y that start in u and end in v. Since our goal is to design a complete algorithm running on a potentially infinite alphabet, we are allowed to compare letters with an equality operator only.

Let $y[i\,..\,j]$ be a run of period p we are looking for, that is, a run that satisfies $i < |u|$ and $j \geq |u|$. Then the run has a full period in u or in v (or in both), which means that at least one of the following conditions holds: $i \leq |u| - p$ and $j \geq |u| + p$.

In the following we deal with runs satisfying the second condition; they

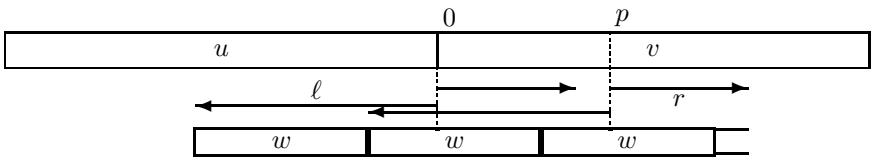


Figure 3.1: Period extensions: $r = \mathrm{Pref}_v[p]$ and $\ell = \mathrm{Pref}_{\tilde{u}\#\tilde{v}\tilde{u}}[|uv| - p + 1]$. A run of period p is detected when $\ell + r \geq p$, which guarantees an exponent at least 2. That means ℓ is the longest common prefix between $\tilde{u}\#\tilde{v}\tilde{u}$ and $\tilde{u}\#\tilde{v}\tilde{u}[|uv| - p + 1\,..\,|\tilde{u}\#\tilde{v}\tilde{u}| - 1]$.

have a full period in v, the other case is symmetric. We consider all possible periods of the runs, $p = 1, 2, \ldots, |v|$, and extend the prefix $v[0\,..\,p-1]$ of v into a factor of period p as much as possible to the right and to the left. This is done with the help of two pre-computed tables, Pref_v and $\mathrm{Pref}_{\tilde{u}\#\tilde{v}\tilde{u}}$.

Recall that, for any nonempty string x, the table Pref_x is defined by

$$\mathrm{Pref}_x[i] = \text{ longest common prefix between } x \text{ and } x[i\,..\,|x| - 1]$$

for $i = 0, \ldots |x| - 1$. In the Section 2 of [45] authors show that Pref_x can be computed in time $O(|x|)$ using only the equality operator to compare letters (see also [13, Section 1.6]).

The end position of a potential run of period p in uv is $j = p + r - 1$ where $r = \mathrm{Pref}_v[p]$. The starting position of that potential run is $i = |u| - \ell$ where ℓ is the maximal length of common suffixes between u and $uv[0\,..\,p-1]$, that is $\ell = \mathrm{Pref}_{\tilde{u}\#\tilde{v}\tilde{u}}[|uv| - p + 1]$. The situation is illustrated in Figure 3.1.

To report the correct period of each run, the algorithm has to avoid non-primitive period strings (i.e. non-primitive $v[0\,..\,p-1]$). To do so, it contains an additional feature: periods are marked when they are multiples of previously found periods of runs and they are not treated when examined later.

RRIP(u, v)

```
 1  P ← Pref_v
 2  S ← Pref_{ũ#ṽũ}
 3  unmark all periods 1, 2, ..., |v|
 4  for p ← 1 to |v| do
 5      if p is not marked then
 6          r ← P[p]
 7          ℓ ← S[|uv| − p + 1]
 8          if ℓ + r ≥ p then
 9              process run u[|u| − ℓ .. |u| − 1]v[0 .. p + r − 1] of period p
10              q ← 2p
11              while q ≤ p + r − 1 do
12                  mark period q
13                  q ← q + p
14  return R
```

Proposition 1. *Algorithm* RRIP *applied to string u and v drawn from an infinite alphabet runs in time $O(|uv|)$.*

Proof. The computation of P at line 1 takes time $O(|u|)$ and can be implemented to run on strings drawn from an infinite alphabet. Similarly, the computation of S at line 2 satisfies the same property and takes time $O(|uv|)$.

Note that a period is marked only once: if it was marked twice, the periodicity lemma applied to the corresponding prefix of v would imply that at least one root of run is not primitive, a contradiction. Therefore, the loop at lines 4–13 executes in time $O(|v|)$, which gives the result. □

Algorithm LRIP, dealing with runs having a full period in u, is designed symmetrically to the above algorithm. It has the same property and runs in the same time.

To detect runs in a string y, a standard balanced divide-and-conquer method using algorithms RRIP and LRIP leads to a solution running in time $O(|y| \log |y|)$. However, this solution is not correct because left and right halves of y cannot

be processed independently. For example, a run discovered in the first half of y may be the prefix of a run that ends in the second half. Additionally, the solution is not satisfactory if each run has to be processed because some runs may be detected several times.

2 Computing all runs of a string

In this section we explain how to compute all runs occurring in y and report them once each in time $O(n \log n)$. The solution is built on the above algorithms and copes with the problems raised at the end of the section.

Let y be divided into two halves $u = y[0\,..\,k-1]$ and $v = y[k\,..\,n-1]$ where $k = \lceil n/2 \rceil + 1$. We use the following observation to create an algorithm based on the divide-and-conquer technique.

Observation 1. *Runs of y are divided into three categories:*

1. *runs that end at or before position $k-1$,*
2. *runs that start at or after position k,*
3. *runs that start at or before position $k-1$ and end after it.*

To check that runs do expand beyond the ends of a given factor we consider the letters preceding its occurrence and following it. To do so, algorithm Runs has parameters a, z and b. It produces runs in azb which do not involve letter a nor letter b. Assuming letters # and \$ do not belong to the alphabet of y we get runs in y by calling Runs$(\#, y, \$)$. Letters a and b are also forwarded to the procedures producing runs in products, RightRunsInProduct and LeftRunsInProduct.

RUNS(a, z, b)

```
1  if |z| ≥ 2 then
2      k ← ⌊|z|/2⌋ + 1
3      R ← RUNS(a, z[0..k−1], z[k])
4      R ← R ∪ RUNS(z[k−1], z[k..|z|−1], b)
5      R ← R ∪ RIGHTRUNSINPRODUCT(a, z, k, b)
6      R ← R ∪ LEFTRUNSINPRODUCT(a, z, k, b)
7      return R
8  else return ∅
```

RIGHTRUNSINPRODUCT(a, z, k, b)

```
1  R ← ∅
2  P ← Pref_{z[k..|z|−1]}
3  S ← Pref_{z#z̃[0..k−1]}
4  unmark all periods 1, 2, ..., |z| − k
5  for p ← 1 to |z| − k do
6      if p is not marked then
7          r ← P[k + p]
8          ℓ ← S[|z| − p + 1]
9          if ℓ + r ≥ p then
10             if ℓ > 0 and p + r ≥ ℓ
                 and (k − ℓ > 0)
                  or ((k − ℓ = 0) and (a ≠ z[k − ℓ + p − 1]))
                 and (k + p + r < |z| − 1)
                  or ((k + p + r = |z| − 1) and (z[k + r] ≠ b)) then
11                 R ← R ∪ {(k − ℓ, k + p + r − 1, p)}
12             q ← 2p
13             while q ≤ p + r − 1 do
14                 mark period q
15                 q ← q + p
16 return R
```

To avoid reporting false runs or run duplicates, the last two algorithms are updated according to the previous Observation 1 on page 20. This is done at line 10:

- Test $\ell > 0$ ensures the detected run, which has a full period in the right half of z, effectively starts in the left half.
- The role of test $p + r \geq \ell$ is to distinguish runs found with RightRunsInProduct from runs found with LeftRunsInProduct. Note the equivalent comparison in LeftRunsInProduct has to be strict ($>$ instead of $\geq$).
- Test $(k - \ell > 0)$ or $((k - \ell = 0)$ and $(a \neq z[k - \ell + p - 1]))$ ensures that the run does not expand to the left of z.
- Test $(k + p + r < |z| - 1)$ or $((k + p + r = |z| - 1)$ and $(z[k + r] \neq b))$ ensures that the run does not expand to the right of z.

As a consequence of the changes, no run is added twice to the output set and we get the following theorem. The running time is deduced from the analysis of the balanced divide-and-conquer technique it uses and from Proposition 1 (page 19).

Theorem 1. *Algorithm* Runs *reports all the runs occurring in its input string* y *in time* $O(|y| \log |y|)$.

3 Computing local periods

The algorithm of the previous section leads to a simple algorithm for computing local periods of strings over infinite alphabets.

Given an alphabet A, recall that an overlap of a pair of strings s and t is a nonempty string w for which

$$A^*s \cap A^*w \neq \emptyset \text{ and } tA^* \cap wA^* \neq \emptyset.$$

The length $|w|$ is a local period of st at position $|s|$. The smallest of these quantities is called the local period of st at position $|s|$ and denoted by $\mathrm{LocPer}_{st}(|s|)$.

In other words, following [18], an integer p is the local period of string y at position i if p is the length of the shortest string w that satisfies one of the four conditions:

1. $y[0\,..\,i-1]$ is a suffix of w and $y[i\,..\,|y|-1]$ is a prefix of w,
2. $y[0\,..\,i-1]$ is a suffix of w and w is a prefix of $y[i\,..\,|y|-1]$,
3. w is a suffix of $y[0\,..\,i-1]$ and $y[i\,..\,|y|-1]$ is a prefix of w,
4. w is a suffix of $y[0\,..\,i-1]$ and w is a prefix of $y[i\,..\,|y|-1]$.

In the first case $p = |w|$ is the (shortest) period of y. Its computation can be done with the Prefix table of y because

$$p = \min\{q \mid 0 \le q < |y| \text{ and } q + \mathrm{Pref}_y(q) = |y|\} \cup \{|y|\}.$$

The second and third cases can be solved using just the prefix table of y and the prefix table of $\tilde{y}$ respectively.

The fourth case occurs when there is a square centered at position i. The occurrence of the square is part of a run and then its detection can be done with the algorithm of the previous section.

The whole algorithm is updated to compute the table LocPer_z instead of reporting runs. The main change, apart from the technical details on the code, is done to the instruction at line 11 of RightRunsInProduct (and at similar line of LeftRunsInProduct). More precisely the instruction is changed to:

```
1  for i ← k − ℓ + p to k + r do
2        LocPer[i] ← min{LocPer[i], p}
```

Theorem 2. *The local periods of a string of length n drawn from an infinite alphabet can be computed optimally in time $O(n \log n)$.*

Proof. The treatments of cases 1, 2 and 3 clearly take linear time each. Discarding the change in the algorithm, case 4 is solved in time $O(n \log n)$.

To evaluate the whole running time we need to count the number of executions of the instruction at line 2, since each run is processed only once at that line. All the updates of LocPer[i] are done with different values of the

period p. It is simple to prove there are a logarithmic number of them (see for example [13] for the number of occurrences of primitively-rooted squares in a string) therefore there are no more than $O(n \log n)$ updates.

Adding the running time of all the steps gives the stated running time.

The optimality comes from the fact that the algorithm, indeed only the part dealing with case 4, can be used to test whether the string contains a square or not. It is known that such a test has an $\Omega(n \log n)$ lower bound (see [45]). Therefore, the algorithm is time optimal. □

Recall that a position i on the string y is called a critical position if the local period at i is the (global) period of y. The factorisation of y into $y[0\,..\,k-1]y[k\,..\,|y|-1]$ is said to be critical. The next statement is another consequence of the above algorithms.

Corollary 1. *The critical factorisations of a string of length n drawn from an infinite alphabet can be computed in time $O(n \log n)$.*

4 Computing runs with DBF

In this section we describe a simple $O(n \log n)$ algorithm for computing the local periods of the word y from its Dictionary of Basic Factors (DBF), see [22, Chapter 7].

We start with two technical definitions: for a set of integers X and an integer k denote

$$k \ominus X \;=\; \{k - x \;:\; x \in X\}, \qquad k \oplus X \;=\; \{k + x \;:\; x \in X\}.$$

To simplify the computations we use the following two lemmas concerning closely overlapping occurrences of factors. Their proofs can be found in [37].

In Lemma 2, there is a introduction of a notation dealing with occurrences. For a factor v of u, $Occ(v, u)$ is defined to be the set of starting positions of all occurrences of v in u.

Lemma 1. *If $M \geq |u|/2$ then $BordersLarger(u, M)$ is a single arithmetic progression.*

Lemma 2. *Assume* $u, v \in \mathcal{BF}(y)$, $|u| = |v| > 1$ *and* $u = u_1u_2$ *and* $v = v_2v_1$, $|u_1| = |u_2| = |v_1| = |v_2|$. *If* $|Occ(u_1, v)| \geq 3$ *and* $|Occ(v_1, u)| \geq 3$ *then* $\mathit{per}(u_1) = \mathit{per}(v_1)$, *that is, the arithmetic progressions* $Occ(u_1, v)$ *and* $Occ(v_1, u)$ *have the same difference.*

The algorithm investigates all interring positions of the word y of length n . For each interposition p it performs at most $O(\log n)$ steps. In each step it decides if there is a local period of length $[2^k \,..\, 2^{k+1}]$. Since we are only interested in computing the minimal local periods, the computation terminates at the first success.

LocalPeriodsUsingDBF(y)

```
1  LocPer[p] ← undefined for p = 0, ..., |y|
2  compute dictionary of basic factors BF(y)
3  preprocess elements of BF(y) for Occ queries
4  for p ← 1 to |y| − 1 do
5      for k ← 0 to ⌊log2 min(p, |y| − p)⌋ do
6          u ← y[p .. p + 2^k − 1]
7          v ← y[p − 2^k .. p − 1]
8          ▷ note that u, v ∈ BF(y)
9          O_u ← 2^{k+1} ⊖ Occ(u, y[p − 2^{k+1} .. p − 1])
10         O_v ← 2^k ⊕ Occ(v, y[p .. p + 2^{k+1} − 1])
11         C ← O_u ∩ O_v
12         if C ≠ ∅ then
13             LocPer[p] ← min C
14             break
15 return LocPer
```

Theorem 3. *The local periods of a string of length* n *drawn from an ordered alphabet can be computed in time* $O(n \log n)$ *using algorithm* LocalPeriodsUsingDBF.

Proof. The first important step of the algorithm is the preprocessing of Occ queries at line 3. This can be done in $O(n \log n)$ time using hashing arrays.

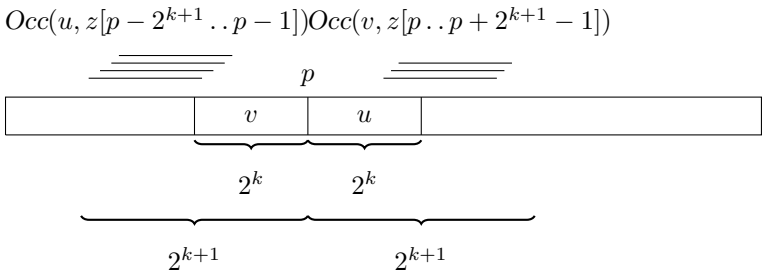


Figure 3.2: Single loop iteration of the algorithm LocalPeriodsUsingDBF.

For each $w \in \mathcal{BF}(z)$ we define $A_{w,i}$ as the arithmetic sequence of occurrences of w starting in $z[i|w| \,..\, (i+1)|w| - 1]$. We can store non-empty entries $A_{w,i}$ in the perfect hash table, such that further queries of the form $Occ(w, z[i \,..\, j])$ (for $j - i = O(|w|)$) can be answered in $O(1)$ time.

The other crucial part of the algorithm is the computation of the intersection of arithmetic sequences $O_u \cap O_v$ at line 11. Luckily, this step can be done in constant time. We can observe that we have two cases:

- One of the sequences is short ($|O_u| < 3$ or $|O_v| < 3$) — so the intersection requires some simple arithmetic operations,
- Both sequences are long ($|O_u| \geq 3$ and $|O_v| \geq 3$) — from Lemma 2 we know that the sequences share the same arithmetic progression. Then once again the computation of the intersection requires only some simple arithmetic operations.

We conclude that each iteration of the loop from line 5 requires only $O(1)$ time. A single iteration of this loop is illustrated on the Figure 3.2. The total time and space complexity of the algorithm is $O(n \log n)$. □

5 Conclusion

Maximal periodicities, called runs, capture consecutive repetitions in strings. We design a direct algorithm to compute them all for a string drawn from an infinite alphabet. In the associated computation model, letter comparisons are

done with an equality operator only. On a string of length n, the algorithm runs optimally in time $O(n \log n)$, although there is a linear number of runs. Under the same hypothesis, we also design a time-optimal algorithm to compute all the local periods of a string, which additionally produces all its critical factorisations. None of the above algorithms depend on an ordering of the alphabet. They show the power of the concept of a prefix table associated with a string for the design of string algorithms. We also design a simple algorithm based on the Dictionary of Basic Factors of the input string. This chapter presents the study idea in runs which is a critical kind of repetitions in the string. However this type of repetitions remains various interesting problems.

4

Gapped palindromes and its anti-exponent

The understanding of palindromes is one of the fundamental problems in language theory and algorithm design. The earlier studies by Manacher [46] and Galil [31], contributed to the construction of linear time algorithms to determine the ordinary palindromes in a string. Crochemore and Rytter [19], presented an algorithm to compute the palindromes of even length in $O(\log n)$ parallel time with n processors. The application of an algorithm created by Knuth, Morris and Pratt [36] shows that the palstar language (the set of concatenations of even palindromes) can be determined in linear time.

In term of molecular biology, palindromic structure has significance to nucleic acid sequence (DNA or RNA) [50, 57]; for example, many restriction enzymes recognize specific palindromic sequences and cut them. Even though the meaning of palindromes is slightly different from the way of genetics since it has to include the nucleotide pairing given by Watson-Crick base pairs. A (single-stranded) nucleotide sequence is said to be a palindrome if it is equal to its reverse complement (Watson-Crick pairing: C is complementary to G while A is complementary to T). For example, the DNA sequence ACCTAGGT is palindromic as its complement is TGGATCCA, and reversing the order of this

complement gives the initial sequence.

Furthermore, palindromes containing a gap (a spacer between left and right replications) have an important role in molecular biology. Such kind of palindromes form a Stem-loop intra molecular base pairing structure (also known as a hairpin or hairpin loop). The occurrence of this structure can be found in single-stranded DNA but more frequently in RNA (see example of genome research [43, 58]), where the structure of the molecule influences its biological function.

In this kind of repetitions, we work on gapped palindromes which are strings of the form $uv\widetilde{u}$ for some strings u, v with $|v| \geq 2$, and where $\widetilde{u}$ denotes the reversal of u. The strings u and $\widetilde{u}$ are named the left arm and the right arm of the palindrome respectively, and v is also known as the gap or the spacer.

Gusfield [33] presented an algorithm for computing gapped palindromes in linear time. However, the algorithm only applies to palindromes of fixed length spacer. Kolpakov and Kucherov [39] extended this work by proposing algorithms for computing two classes of gapped palindromes. The first class requires the condition that $|v| \leq |u|$ and is called long-armed palindromes. While the second is called length-constrained palindromes and defined by bounded spacer length and a lower bound on the arm length, i.e. $MinGap \leq |v| \leq MaxGap$ and $MinLen \leq |u|$, for some constants $MinGap$, $MaxGap$, and $MinLen$. Both classes meet the maximality condition of gapped palindromes. The presented algorithms apply to both classes and run in time $O(n + S)$, where n is the length of the input word and S is the number of output palindromes. However, the combinatorial enumeration of such gapped palindromes is yet unexplored.

Furthermore, an algorithm to find the longest previous reverse factor occurring at each position of string is presented in [8]. This is a principal notion used for optimal detection of various types of palindromes, and leads to the reverse Lempel-Ziv factorisation used in [39]. Let the length of x be n, the LPrF table for x is defined as follows: for i, $0 \leq i < n$, LPrF[i] is the length of the longest prefix of $x[i \,..\, n-1]$ whose reverse occurs in $x[0 \,..\, i-1]$. For each position i such longest prefix is called the Longest Previous reverse Factor at position i on x.

The result on LPrF table computation is obtained by two alternative algorithms: the first one is based on Suffix Tree of the input string while the second relies on its Suffix Automaton. Both algorithms run in linear time on a fixed size alphabet. Typical applications of these algorithms are for RNA secondary structure prediction and for text compression when reverse factors are also accounted for as it is in [32] and subsequent works.

In this chapter, we consider a fixed palindrome-free string, that is, a string containing no palindrome of length greater than 1. For such string, we present a linear-time algorithm to compute the maximal anti-exponent of the gapped palindromes (a preliminary version was presented in [4]). This notion encompasses the detection of the most significant gapped palindromes occurring in a string and can be extended easily to biological palindromes.

The solution proposed here is a special type of divide-and-conquer technique. The technique we use is unbalanced contrary to what is traditional to impose for improving the running time or the memory space of resulting recursive algorithms. In fact, the balanced divide-and-conquer approach is unlikely to improve the running time of our solution as it would lead to a $O(n \log n)$-time algorithm. Our technique is essentially based on the reversed Ziv-Lempel factorisation of the input string, in which factors have various lengths. Despite the unbalanced feature, the solution provides an algorithm running in linear time, at least on a fixed-sized alphabet. This strategy has been initiated in [11] and applied since then to a variety of problems related to repeats occurring in strings, like in [2].

1 Gapped palindromes

Let x be a string of length n drawn from a finite alphabet A. The reversal of x is denoted by $\widetilde{x} = x[n-1]x[n-2]\cdots x[0]$. If $x = \widetilde{x}$, then x is a palindrome.

A factor w of x is a **gapped palindrome** if it is in the form $uv\widetilde{u}$ for some u, v in A^n with $|v| \geq 2$.

Then u and $\tilde{u}$ are defined to be the **anti-borders** of w if and only if u is the longest prefix of w for which $\widetilde{u}$ is a suffix of w. Replicating the standard notion

of string exponent, we define the **anti-exponent** of a gapped palindrome w to be $|w|/|uv|$.

A gapped palindrome is said to be maximal, **a maximal gapped palindrome**, if its both anti-borders cannot be extended outward or inward preserving the palindrome structure. In the other word, if $w = uv\tilde{u}$ is a maximal gapped palindrome, then $w[|u|] \neq w[|uv| - 1]$ and if $\mathbf{c}uv\tilde{u}\mathbf{d}$ is defined for some letters c and d from the alphabet set, then $c \neq d$, as shown in the Figure 4.1.

It is clear that maximal gapped palindrome is not a palindrome as the gap v is not a palindrome, consequently the length of its gap; $|v|$, is at least 2. Moreover, a gapped palindrome in x is said to be a **maximal anti-exponent gapped palindrome**, a **MAXGP** for short, if its anti-exponent is maximal among all gapped palindromes occurring in x.

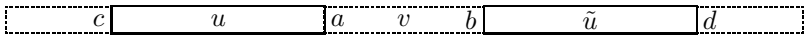


Figure 4.1: Both anti-borders cannot be extended inward or outward ($c \neq d$) preserving palindrome structure, the first and the last letters of v are different ($a \neq b$).

A **palindrome-free string** contains no gapped palindrome of anti-exponent larger than 2, that is, no gapped palindrome of the form $u\alpha\tilde{u}$ for any letter α.

We consider a palindrome-free string x of length n (containing no ordinary palindrome of length larger than 1) and compute the maximal anti-exponent over all its maximal gapped palindromes.

2 Algorithm scheme

The algorithm MaxAntiExpGP is initially presented here, it computes the maximal anti-exponent of gapped palindromes of a palindrome-free string x.

The algorithm detects and processes gapped palindromes of the form $uv\tilde{u}$, for two strings u and v, where u is the longest anti-border of the gapped

palindrome and $|v| \geq 2$.

This is achieved with the help of algorithm MaxAntiExp; explained in the next section, which detects those gapped palindromes occurring in the concatenation of two strings and whose anti-exponent are at least of the current maximal anti-exponent.

Algorithm MaxAntiExpGP relies on the reversed Lempel-Ziv factorisation of a string x. We modify the reversed factorisation for the purpose of our problem by defining z_1 as the longest prefix of x in which no letter occurs more than once.

Algorithm MaxAntiExpGP deals with the phrases z_2 to z_k sequentially. Therefore, at each step, after $z_1, z_2, \ldots, z_{i-1}$ have been processed, the variable $\tilde{e}$ stores the maximal anti-exponent of gapped palindromes of $z_1 z_2 \cdots z_{i-1}$. Then, the next gapped palindromes to be considered are those involving the phrase z_i. These gapped palindromes $uv\tilde{u}$ are either internal to z_i or occur partially in z_i. According to the properties of the reverse factorisation, the occurrence of $\tilde{u}$ is only to be searched for within $z_{i-1}z_i$. The reason is that $\tilde{u}$ cannot contain a phrase as this would contradict the definition of the reverse factorisation.

We further distinguish four possible cases according to the position of the gapped palindrome $uv\tilde{u}$ as follows (see Figure 4.2):

(i) Both occurrences of u and $\tilde{u}$ are contained in z_i.

(ii) The occurrence of u is contained in z_{i-1}, while $\tilde{u}$ ends in z_i.

(iii) The occurrence of u starts in z_{i-1}, while $\tilde{u}$ is contained in z_i.

(iv) The occurrence of u starts in $z_1 \cdots z_{i-2}$, while $\tilde{u}$ is contained in $z_{i-1}z_i$.

In Case (i), the gapped palindrome $uv\tilde{u}$ contained in z_i might be different from the one appearing before, however, they have the same anti-exponent, so this case needs no further action. The other cases are handled by calls to algorithm MaxAntiExp as described in the following section. For any two strings z and w, and a positive rational number $\tilde{e}$, MaxAntiExp$(z, w, \tilde{e})$ is the maximal anti-exponent of gapped palindromes occurring within zw, whose anti-exponent is at least $\tilde{e}$; this value remains $\tilde{e}$ if there is no such gapped palindrome.

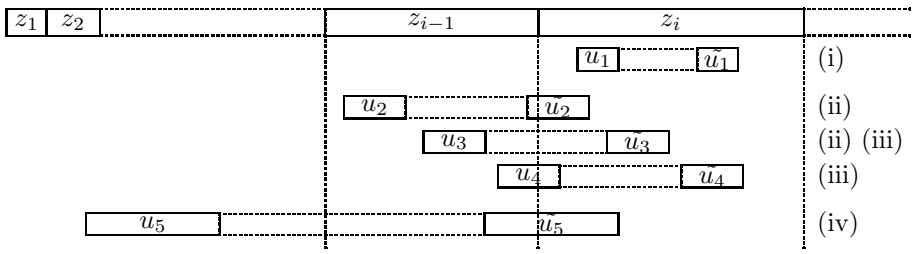


Figure 4.2: The only four possible positions of a gapped palindrome $uv\tilde{u}$ involving phrase z_i of the reversed factorisation of the string: (i) both u and $\tilde{u}$ are internal to z_i; (ii) occurrence of u is internal to z_{i-1} and $\tilde{u}$ ends in z_i; (iii) occurrence of $\tilde{u}$ is internal to z_i and occurrence of $\tilde{u}$ is in z_i; (iv) the occurrence of u starts in $z_1 \cdots z_{i-2}$ and of $\tilde{u}$ is internal to $z_{i-1}z_i$.

MaxAntiExpGP(x)

1 $(z_1, z_2, \ldots, z_k) \leftarrow$ reversed-factorisation of x
2 $\triangleright$ z_1 is the longest prefix of x in which no letter repeats
3 $\tilde{e} \leftarrow 1$
4 **for** $i \leftarrow 2$ **to** k **do**
5 $\quad \tilde{e} \leftarrow max\{\tilde{e}, \text{MaxAntiExp}(z_{i-1}, z_i, \tilde{e})\}$ (for case (ii))
6 $\quad \tilde{e} \leftarrow max\{\tilde{e}, \text{MaxAntiExp}(\widetilde{z_i}, \widetilde{z_{i-1}}, \tilde{e})\}$ (for case (iii))
7 $\quad$ **if** $i > 2$ **then**
8 $\quad\quad \tilde{e} \leftarrow max\{\tilde{e}, \text{MaxAntiExp}(\widetilde{z_{i-1}z_i}, z_1\widetilde{\cdots z_{i-2}}, \tilde{e})\}$(for case (iv))
9 **return** $\tilde{e}$

Notice that both $z_1\widetilde{\cdots z_{i-2}}$ and $\widetilde{z}_{i-2} \cdots \widetilde{z}_1$ mean the reversal of $z_1 \cdots z_{i-2}$ however $z_1\widetilde{\cdots z_{i-2}}$ can be claimed to case (iv) in more sensible way. Another notice, variable $\tilde{e}$ can be initialised to $(|A|+1)/|A|$, when A is the alphabet of the input string x and if x is long enough (see the following remark).

Remark 1. *When A is the alphabet set of the input string x and $|x| > |A|$, let y be a substring of x which is composed by one appearance of all characters from the alphabet set, hence $|y| = |A|$. If a is a character from A being adjacent to y and the first letter of y is also a, then factor ya is a gapped palindrome. The anti-exponent of this worst gapped palindrome is $(|y|+1)/|y|$. Then variable $\tilde{e}$*

can be initialised to $(|A|+1)/|A|$.

Theorem 4. *For any given palindrome-free string,* MaxAntiExpGP *computes the maximal anti-exponent of gapped palindromes in the string.*

Proof. Recall that the maximal anti-exponent of any palindrome-free string is no less than 1, then $\tilde{e}$ is initialised to 1. Let $\tilde{e}_2, \tilde{e}_3, \ldots, \tilde{e}_k$ be the successive values of the variable $\tilde{e}$, where $\tilde{e}_i$ is the value of $\tilde{e}$ just after the execution of lines 5–8 for index i.

Then we show that $\tilde{e}_i$ is the maximal anti-exponent of gapped palindromes occurring in $z_1 z_2 \cdots z_i$ when $\tilde{e}_{i-1}$ is that of $z_1 z_2 \cdots z_{i-1}$, for $2 \leq i \leq k$.

We only have to consider gapped palindromes occurring in the concatenation of $z_1 z_2 \cdots z_{i-1}$ and z_i, that is, gapped palindromes of the form $uv\tilde{u}$ where u occurs in $z_1 z_2 \cdots z_{i-1}$ and $\tilde{u}$ ends in z_i. As discussed above and illustrated in Figure 4.2, only four cases are to be considered, because $\tilde{u}$ cannot start in $z_1 z_2 \cdots z_{i-2}$, otherwise it contradicts the definition of z_{i-1}.

Line 5 deals with Case (ii) by the definition of MaxAntiExp. Similarly, line 6 is for Case (iii), and line 8 is for Case (iv). However, we do not need to deal with Case (i) as if a gapped palindrome occurs entirely in z_i, by the definition of z_i, then its reverse must occur previously in $z_1 z_2 \cdots z_{i-1}$; which is already reported by $\tilde{e}_{i-1}$.

Thus, all relevant gapped palindromes are considered in the computation of $\tilde{e}_i$, which is then the maximal anti-exponent of gapped palindromes occurring in $z_1 z_2 \cdots z_i$. This implies that the anti-exponent $\tilde{e}_k$, returned by the algorithm, is that of $z_1 z_2 \cdots z_k = x$; which completes the proof. □

3 Computing the maximal anti-exponent of gapped palindrome

In the previous section we had a minor introduction of algorithm MaxAnti-Exp, hence in this section we go into detail. The algorithm MaxAntiExp is designed to compute the maximal anti-exponent of gapped palindromes, $uv\widetilde{u}$, occurring in the product of two strings: z and w.

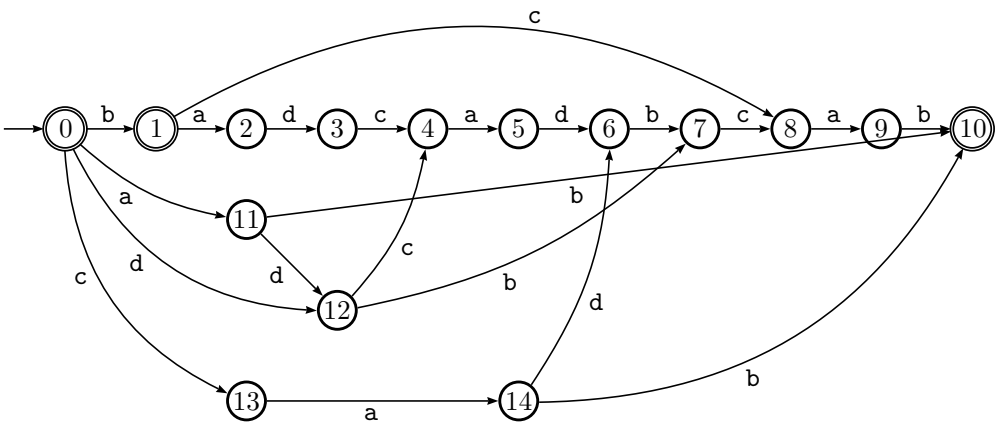


Figure 4.3: Suffix Automaton of $w =$ `badcadbcab`.Suffix links: $F_w[1] = 0$, $F_w[2] = 11$, $F_w[3] = 12$, $F_w[4] = 13$, $F_w[5] = 14$, $F_w[6] = 12$, $F_w[7] = 1$, $F_w[8] = 13$, $F_w[9] = 14$, $F_w[10] = 1$, $F_w[11] = 0$, $F_w[12] = 0$, $F_w[13] = 0$, $F_w[14] = 11$. Maximal incoming string lengths: $L_w[0] = 0$, $L_w[1] = 1$, $L_w[2] = 2$, $L_w[3] = 3$, $L_w[4] = 4$, $L_w[5] = 5$, $L_w[6] = 6$, $L_w[7] = 7$, $L_w[8] = 8$, $L_w[9] = 9$, $L_w[10] = 10$, $L_w[11] = 1$, $L_w[12] = 2$, $L_w[13] = 1$, $L_w[14] = 2$. Shortest Prefix lengths: $sp_w[0] = 0$, $sp_w[1] = 1$, $sp_w[2] = 2$, $sp_w[3] = 3$, $sp_w[4] = 4$, $sp_w[5] = 5$, $sp_w[6] = 6$, $sp_w[7] = 7$, $sp_w[8] = 8$, $sp_w[9] = 9$, $sp_w[10] = 10$, $sp_w[11] = 2$, $sp_w[12] = 3$, $sp_w[13] = 4$, $sp_w[14] = 5$.

The algorithm considers gapped palindromes $uv\widetilde{u}$ where left anti-border u starts in z, right anti-border $\widetilde{u}$ ends in w and whose anti-exponent is at least $\tilde{e}$.

In order to locate these gapped palindromes, the algorithm calculates for each position j on z, the longest potential anti-border u of a gapped palindrome $uv\tilde{u}$; the longest prefix of $z[j..|z|-1]w$, whose reverse $\tilde{u}$ ends in w. The algorithm is built upon an algorithm that finds all gapped palindromes using the Suffix Automaton of string w.

The Suffix Automaton of w, denoted $\mathcal{S}(w)$, is the minimal deterministic finite automaton whose language is the set of suffixes of w, an example is given in Figure 4.3.

The suffix Automaton $\mathcal{S}(w)$ can be used to compute the factor r, which is

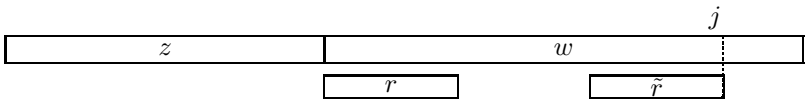


Figure 4.4: Factor r is the longest prefix of w whose reverse occurs in w, where r and $\widetilde{r}$ do not overlap and position j is the end position of $\widetilde{r}$.

the longest prefix of w whose reverse occurs in w, such that r and $\widetilde{r}$ do not overlap (See Figure 4.4 for factor r.) This factor can be computed by reading $\tilde{w}$ along the main line of the suffix automaton of w. Algorithm MaxAntiExp then aims to extend this factor r to the left and $\widetilde{r}$ to the right; this is achieved by reading z backward and exploiting the Suffix Automaton of w.

Before detailing MaxAntiExp algorithm, we present the following two lemmas. These lemmas show that, after u is located (by extending r to the left), and although some of its prefixes may have anti-exponents higher than $\tilde{e}$, we can discard many of them.

Lemma 3. *Let u' be a prefix of u. If both of their reverses $\tilde{u'}$ and $\tilde{u}$ associate with the same state of $\mathcal{S}(w)$ the maximal anti-exponent of a $u'v'\tilde{u'}$ is not greater than the maximal anti-exponent of its associated $uv\tilde{u}$ gapped palindrome.*

Proof. The hypothesis implies that the right-most occurrence of u' starts at the same positions on z as u (see Figure 4.5). Then, $u'v'\tilde{u'}$ and $uv\tilde{u}$ have the same length $|uv\tilde{u}| = |u'v'\tilde{u'}|$ but since u' is no longer than u, the anti-exponent of $u'v'\tilde{u'}$ is not greater than that of $uv\tilde{u}$. □

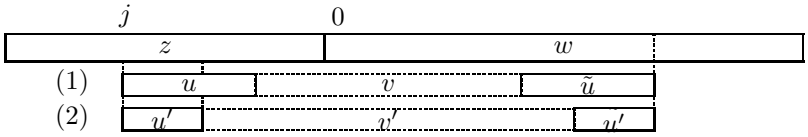


Figure 4.5: When $\tilde{u}$ and its suffix $\tilde{u'}$ end at the same right-most position on w, gapped palindrome (1) has a larger exponent than gapped palindrome (2).

Note that a prefix u' of u may have an internal occurrence in $uv\tilde{u}$, which would lead to a gapped palindrome having a larger anti-exponent. For example, let $z = \texttt{bbacbbd}$ and $w = \texttt{cdabb}$. The gapped palindrome `bbacbbdcdabb` with anti-border `bba` has anti-exponent 12/9 while the prefix `bb` of `bba` infers the gapped palindrome `bbdcdabb` of greater anti-exponent 8/6.

Lemma 4. *If u occurs at positions j and k on z with $k > j$, the gapped palindrome $uv'\tilde{u}$ starting at j cannot be a maximal anti-exponent gapped palindrome.*

Proof. To have a maximal anti-exponent the anti-border u in $uv'\tilde{u}$ should occur at the right-most position on z. Otherwise there is a gapped palindrome sharing the same right anti-border $\tilde{u}$ and with a closer u (see Figure 4.6). Therefore, $1 + |u|/|uv| > 1 + |u|/|uv'|$, which completes the proof. □

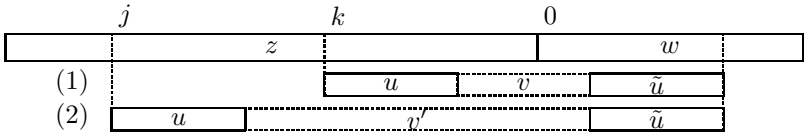


Figure 4.6: Gapped palindrome (1) starting at position k has a larger anti-exponent than gapped palindrome (2) starting at position $j < k$.

The properties stated in the previous lemmas are used by algorithm MaxAntiExp to avoid some anti-exponent calculations as follows: The algorithm proceeds on z backwards starting from the last position of z. At position j, let $uv\tilde{u}$ be a gapped palindrome starting at j on $z[j\,..\,|z|-1]w$ and for which $\tilde{u}$ is the longest string associated with state $q = \text{goto}(\mathit{initial}(\mathcal{S}), \tilde{u})$, where goto is the transition function of the automaton. Then next occurrences of u and of any of its prefixes cannot produce gapped palindromes with an anti-exponent larger than that of $uv\tilde{u}$. State q is then marked to inform the next steps of the algorithm that it has been visited.

We modify the Suffix Automaton $\mathcal{S}(w)$ to include yet another useful function, sp_w, defined on states of $\mathcal{S}(w)$ as follows:

$sp_w[p]$ is the minimal length of paths from initial state to p; in other terms, if $p = \text{goto}(\text{initial}(\mathcal{S}(w)), x)$, then $sp_w[p] = |x'x|$ where x' is the shortest string for

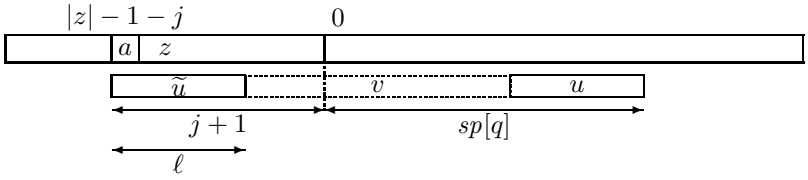


Figure 4.7: The anti-exponent of all gapped palindromes having u as anti-border, can be computed by $(j + sp[q])/(j - \ell + sp[q])$.

which $x'x$ is a prefix of w. With use of these additional precomputed elements, the value of the anti-exponent can be simply determined (see Figure 4.7).

MaxAntiExp$(z, w, \tilde{e})$

1 $\mathcal{S} \leftarrow$ *Suffix Automaton of* w
2 $(q, \ell) \leftarrow$ Read$(initial(S), \tilde{w})$
3 $\triangleright$ Read() returns goto$(initial(S), \tilde{r})$ and $|r|$ (See Figure 4.4).
4 mark initial$(\mathcal{S})$
5 **for** $j \leftarrow 0$ **to** $\min\{\lfloor |w|/(\tilde{e}-1)-1 \rfloor, |z|-1\}$ **do**
6 **while** goto$(q, z[|z|-1-j]) =$ NIL **and** $q \neq$ initial$(\mathcal{S})$ **do**
7 $(q, \ell) \leftarrow (F[q], L[F[q]])$
8 **if** goto$(q, z[|z|-1-j]) \neq$ NIL **then**
9 $(q, \ell) \leftarrow (\text{goto}(q, z[|z|-1-j]), \ell+1)$
10 $(q', \ell') \leftarrow (q, \ell)$
11 **while** q' unmarked **do**
12 $\tilde{e} \leftarrow \max\{\tilde{e}, (j+1+sp[q])/(j+1-\ell+sp[q])\}$
13 **if** $\ell' = L[q']$ **then**
14 mark q'
15 $(q', \ell') \leftarrow (F[q'], L[F[q']])$
16 **return** $\tilde{e}$

Figure 4.8 illustrates a computation done by the algorithm MaxAntiExp using the Suffix Automaton of Figure 4.3.

j		0	1	2	3	4	5	6	7	8	9	10
$z[\|z\|-1-j]$		c	a	d	b	c	a	b	d	a	c	b
q	10	8	9	6	7	8	9	10	12	11	13	1
ℓ	2	2	3	3	4	5	6	7	1	1	1	1
anti-exp		$\frac{9}{7}$	$\frac{11}{8}$	$\frac{9}{6}$	$\frac{11}{7}$	$\frac{13}{8}$	$\frac{15}{9}$	$\frac{17}{10}$	$\frac{11}{10}$	$\frac{11}{10}$	$\frac{14}{13}$	$\frac{12}{11}$
		$\frac{5}{4}$	$\frac{7}{5}$	$\frac{6}{4}$	$\frac{7}{6}$							

Figure 4.8: For $z =$ `bcadbacbdac` and $w =$ `badcadbcab`, computing anti-exponents when searching the zw for gapped palindromes $uv\widetilde{u}$ is shown above. The left anti-border begins in z and its reversal; $\widetilde{z}$ occurs in w. The Suffix Automaton of w with function sp is in Figure 4.3. The search is done by parsing z backward with the automaton. Anti-exponents of gapped palindromes can be determined by the expression $(j+1+sp[q])/(j+1-\ell+sp[q])$. The maximal anti-exponent among all gapped palindromes is 17/10.

Theorem 5. *Algorithm* MaxAntiExp*, applied to strings z, w, and to the rational number $\tilde{e}$, computes the maximal anti-exponent of gapped palindromes in zw, whose occurrence of its left anti-border starts in z, while its right reversal anti-border occurs in w and its anti-exponent is at least $\tilde{e}$.*

Proof. In the Algorithm MaxAntiExp, the position $|z|-1-j$ on z indicates a potential starting position of a gapped palindrome. First, we show that the algorithm does not require to investigate more values of j than those specified at line 4.

We know that anti-exponent of a gapped palindrome $uv\tilde{u}$ is $uv\tilde{u}/uv$. Furthermore, the only significant gapped palindromes are the ones whose anti-exponent is greater than $\tilde{e}$. In addition, we know the longest possible gapped palindrome contained in zw has period $j+1$ and border w. Then $(|w|+j+1)/j+1 > \tilde{e}$ implies $j < |w|/(\tilde{e}-1)$. Since $|z|-1-j$ is a position on z, j must be less than $|z|$, that concludes the first claim.

For any integer j, we show that the algorithm searches into all the possible relevant gapped palindromes having an anti-exponent of at least $\tilde{e}$ and

beginning at position $|z| - j - 1$. The following property related to variables q, states of $\mathcal{S}$ and ℓ is known from [13, Section 6.6]: let u be the longest prefix of $z[|z|-1-j\,..\,|z|-1]w$ whose reverse is a factor of w, then $q = \text{goto}(\text{initial}(\mathcal{S}), \tilde{u})$ and $\ell = |\tilde{u}| = |u|$. We have r, the longest prefix of w whose reverse factor appears in w, hence the property is as well hold after executing line 2 for only string w because of the initial value of the variables q and ℓ taken from the benefit of factor r.

Then string u is the left anti-border of a gapped palindrome occurring in z whose right reversal border, $\tilde{u}$ ends in w. Lines 9 to 14 check the anti-exponents associated with $\tilde{u}$ and its longest prefixes leading to a different state of $\mathcal{S}$: If q' is unmarked, the exponent is computed as explained before (see Figure 4.7). If the condition at line 11 is met, which means that u is the longest word satisfying $q' = \text{goto}(\text{initial}(\mathcal{S}), \tilde{u})$. Due to Lemma 6 the algorithm does not need to check the exponent associated with later occurrences of u, nor with the suffixes of u since they have been checked before. Due to Lemma 5, suffixes of u ending at the same right-most position on z do not have a larger exponent. Therefore the next suffix whose associated exponent has to be checked is the longest suffix leading to a different state of $\mathcal{S}$: it is $F(q')$ and the length of the suffix is $L(F(q'))$ by definition of F and L.

Finally note the initial state of $\mathcal{S}$ is marked because it corresponds to an empty string u, that is a gapped palindrome of exponent 1, which is not larger than the values of $\tilde{e}$.

This proves that the algorithm runs through each possible relevant gapped palindrome whose occurrence of left border is in z, while its right reversal border ends in w. □

4 Complexity analysis

In this section, the running time and space requirements of created algorithms are analysed.

Proposition 2. *Applied to strings z and w and to the rational number $\tilde{e}$, Algorithm* MaxAntiExp *requires $O(|w|)$ space in addition to inputs and runs*

in total time $O(|w| + \min\{\lfloor |w|/(\tilde{e}-1) - 1 \rfloor, |z| - 1\})$ *on a fixed size alphabet. It performs less than* $2|w| + \min\{\lfloor |w|/(\tilde{e}-1) - 1 \rfloor, |z| - 1\}$ *anti-exponent computations.*

Proof. The space is used mostly for storing the automaton, which is known to have no more $2|w|$ states and $3|w|$ edges (see [13]). It can be stored in linear space if edges are implemented by successor lists, which adds a multiplicative $\log a$ factor on transition time, where a is the alphabet size.

Including the automaton construction with elements F, L and sp, it is known from [13, Section 6.6] that the algorithm (excluding lines 9 to 14) runs in linear time on a fixed alphabet.

Next, let we consider the number of times line 11 is executed, that is done once for each j associated with an unmarked state. If it is done more than once for a given position, then the second value of q' comes from the failure link. A crucial observation is that condition at line 12 holds for such a state. Therefore, since $\mathcal{S}(z)$ has no more than $2|w|$ states, the total number of extra executions of line 11 is at most $2|w|$, which leads to the result as stated. □

The proof of the linear running time of Algorithm MaxAntiExpGP additionally relies on a combinatorial property of strings. If x is the input string from alphabet size a, the maximal anti-exponent unavoidable in x is $(a+1)/a$ (See Remark 2).

Theorem 6. *Applied to any palindrome-free string of length* n *on a fixed-size alphabet, Algorithm* MaxAntiExpGP *runs in time* $O(n)$ *and requires* $O(n)$ *extra space.*

Proof. Computing the reversed factorisation $(z_1, z_2, \ldots, z_k)$ of the input string takes time and space $O(n)$ on a fixed-size alphabet using any suffix data structure, see [20].

Next instructions execute in linear extra space from Proposition 3. Line 5 takes time $O(|z_i| + \min\{\lfloor |z_i|/(\tilde{e}-1) + 1 \rfloor, |z_{i-1}| - 1\})$, which is bounded by $O(|z_i| + |z_i|/(\tilde{e}-1) - 1)$, for $i = 2, \ldots, k$. As we detailed above, $\tilde{e}$ is at least $(a+1)/a$ where a is the size of the input alphabet. The time is then bounded

by $O(|z_i| + |z_i|/(((a+1)/a) - 1) + 1)$, then $O(|z_i|)$ as $(a+1)/a > 1$ for long enough input. So the total running time of Line 5 is $O(\Sigma_{i=2}^{k}|z_i|)$.

Similarly, it is $O(\Sigma_{i=2}^{k}|z_{i-1}|)$ for Line 6 and $O(\Sigma_{i=2}^{k}|z_{i-1}z_i|)$ for Line 8. Thus the overall runtime is bounded by $O(\Sigma_{i=1}^{k}|z_i|)$, which is $O(n)$. □

5 Conclusion

As a consequence of this work, algorithm MaxAntiExpGP calculates the maximal anti-exponent of gapped palindromes occurring in an input palindrome-free string. It makes calls to algorithms MaxAntiExp which is designed to compute the maximal anti-exponent of gapped palindromes occurring in a concatenation of two strings.

Both algorithms run in linear-time on a fixed-size alphabet, the running time of the main algorithm MaxAntiExpGP is proportional to the size of its input.

However, as far as we know, the number of distinct gapped palindromes in a string x whose anti-exponents equals to the maximal anti-exponent of x is unknown and constitutes an interesting combinatoric problem.

Another interesting question is the notion of a smallest unavoidable anti-exponent that we call the *anti-repetitive threshold* of the alphabet. If $\tilde{e}$ is this anti-exponent, then it is the smallest rational number for which there exists an infinite string whose finite gapped palindromes have at most $\tilde{e}$ anti-exponents.

Dejean [25] introduced similar notion for factor exponents and referred to it as the repetitive threshold $\mathrm{RT}(\sigma)$ of an alphabet of size σ. It is the smallest rational number for which there exists an infinite string whose finite factors have exponent at most $\mathrm{RT}(\sigma)$. It is known from Thue [55] that $RT(2) = 2$, Dejean [25] proved that $RT(3) = 7/4$ and stated the exact values of $\mathrm{RT}(\sigma)$ for every alphabet size $\sigma > 3$. Dejean's conjecture was eventually proved in 2009 after partial proofs given by several authors (see [49, 24] and ref. therein).

In addition to the algorithmic aspect of the study of gapped palindromes, this work opens a new research subject in Combinatorics on Words.

5

Repeats and its exponent

In this chapter, we consider the question of computing the maximal exponent of factors (substrings) of a given string. Repeats considered in this work are of exponent at most 2. They refer to strings of the form uvu where u is its longest border (both a prefix and a suffix). The study of repeats in a string has to do with long-distance interactions between separated occurrences of the same segment (the u part) in the string. Although occurrences may be far away from each others, they may interact when the string is folded as it is the case for genomic sequences. The notion of maximal exponent is central in questions related to the avoidability of patterns in infinite words. An infinite word is said to avoid e-powers (resp. e^+-powers) if the exponents of its finite factors are smaller than e (resp. no more than e).

Dejean [25] introduced the repetitive threshold $\mathrm{RT}(a)$ of an a-letter alphabet. It is the smallest rational number for which there exists an infinite word whose finite factors have exponent at most $\mathrm{RT}(a)$. In other words, the maximal exponent of factors of such a word is $\mathrm{RT}(a)$, the minimum possible. The word is also said to be $\mathrm{RT}(a)^+$-power free. It is known from Thue [55] that $r(2) = 2$, Dejean [25] proved that $r(3) = 7/4$ and stated the exact values of $\mathrm{RT}(a)$ for every alphabet size $a > 3$. Her conjecture was eventually proved in 2009 after partial proofs given by several authors (see [49, 24] and ref. therein).

The exponent of a string can be calculated in linear time using basic string matching that computes the smallest period associated with the longest border of the string (see [13]). A straightforward consequence provides a $O(n^3)$-time solution to compute the maximal exponent of all factors of a string of length n since there are potentially of the order of n^2 factors. However, a quadratic time solution is also a simple application of basic string matching. By contrast, our solution runs in linear time on a fixed-size alphabet.

When a string contains runs, that is, maximal occurrences of repetitions of exponent at least 2, computing their maximal exponent can be done in linear time by adapting the algorithm of Kolpakov and Kucherov [38] that computes all the runs occurring in the string. Their result relies on the fact there exists a linear number of runs in a string [38] (see [52, 15] for precise bounds). Nevertheless, this does not apply to square-free strings, which we are considering here.

Our solution works indeed on overlap-free strings for which the maximal exponent of factors is at most 2. Thus, we are looking for factors w of the form uvu, called repeats, where u is the longest border of w. To do so, we use two main tools: a factorisation of the string and the Suffix Automaton of some factors.

The Suffix Automaton is used to search for maximal repeats in a product of two strings due to its ability to locate occurrences of all factors of a pattern. Here, we enhance the automaton to report the right-most occurrences of those factors. Using it solely in a balanced divide-and-conquer manner produces a $O(n \log n)$-time algorithm. To remove the log factor we additionally exploit a string factorisation, namely the f-factorisation, a type of LZ77-factorisation (see [56]) fit for string algorithms. It has now become common to use it to derive efficient or even optimal algorithms. The f-factorisation allows to skip larger and larger parts of the strings during an online computation. For our purpose, it is composed of factors occurring before their current position with no overlap. The factorisation can be computed in $O(n \log a)$-time using a Suffix Tree or a Suffix Automaton, and in linear time on an integer alphabet using a Suffix Array [23].

The running time of the proposed algorithm depends additionally on the repetitive threshold of the underlying alphabet size of the string. The threshold restricts the context of the search for a second occurrence of u associated with a repeat uvu.

We show a very surprising property of repeats whose exponent is maximal in an overlap-free string: there are no more than a linear number of occurrences of them, although the number of occurrences of maximal (i.e. non-extensible) repeats can be quadratic. As a consequence, the core algorithm can be modified to output all occurrences of maximal-exponent repeats of an overlap-free string in linear time.

The question would have a simple solution by computing MinGap on each internal node of the Suffix Tree of the input string, as is discussed in the conclusion. MinGap of a node is the smallest difference between the positions assigned to leaves of the subtree rooted at the node. Unfortunately, the best algorithms for MinGap computation, equivalent to MaxGap computation, run in time $O(n \log n)$ (see [5, 35, 7]) and the discussion in [9]).

A remaining question to the present study is to unify the algorithmic approaches for repetitions (exponent ≥ 2) and for repeats (exponent ≤ 2).

The supporting paper for this chapter can found in [3].

1 Maximal-exponent repeats

An overlap-free string is a string containing no factor of exponent larger than 2, that is, no factor of the form $bwbwb$ for a letter b and a string w. As mentioned in Theoretical Background, repeat is a string of exponent at most 2. (A repetition is usually a string of exponent at least 2.)

In this chapter, we consider an overlap-free string y of length n on a finite alphabet (a fixed overlap-free string) and deal with the repeats occurring in it. A repeat w in y is a factor of the form uvu. We often consider the decomposition uvu for which u is the longest border of w (longest factor that is both a prefix and a suffix of w). Then $\text{period}(w) = |uv|$ and $\exp(w) = |uvu|/|uv| = 1 + |u|/\text{period}(w)$. By convention, in the following we allow a border-free factor

to be considered as a repeat of exponent 1, though this is not a repeat in the common sense since the repeating element u is empty, i.e. does not exist.

A repeat in y is said to be a **maximal-exponent repeat**, a MER for short, if its exponent is maximal among all repeats occurring in y. An occurrence of a repeat is said to be a maximal, a **maximal repeat** for short and abuse of terms, if it cannot be extended to the left nor to the right preserving the same period. That mean if $r = uvu$ is a maximal repeat in y and $\mathbf{a}uvu\mathbf{b}$ is defined for some letters a and b from the alphabet set, then $a \neq r[|uv| - 1]$ and $r[|u|] \neq b$.

Note all occurrences of any MER is a maximal repeat but the converse is obviously false.

2 Computing the maximal exponent of repeats

The core result of the chapter is the algorithm, MaxExpRep, that computes the maximal exponent of factors of the overlap-free string y. The algorithm searches for factors that are repeats of the form uvu, for two strings u and v, with u being the longest border of the repeat. This is achieved with the help of Algorithm MaxExp, designed in the next section, which detects those repeats occurring when two strings are concatenated.

Algorithm MaxExpRep relies on the adaptation of the f-factorisation of the string y. We apply the factorisation to the purpose of our problem by defining z_1 as the longest prefix of y in which no letter occurs more than once. Then, $|z_1| \leq a$ and $\text{MaxExpRep}(z_1) = 1$. Note that $\text{MaxExpRep}(z_1 z_2) > 1$ if $z_1 \neq y$.

As the factorisation of y is computed, Algorithm MaxExpRep processes the phrases sequentially, from z_2 to z_k. After z_1, z_2, ..., z_{i-1} have been processed, the variable e stores the maximal exponent of factors of $z_1 z_2 \cdots z_{i-1}$. Then, the next repeats to be considered are those involving phrase z_i. Such a repeat uvu can either be internal to z_i or involve other phrases. But the crucial property of the factorisation is that the second occurrence of u is only to be searched for in $z_{i-1} z_i$ because it cannot contain a phrase as this would contradict the definition of the factorisation.

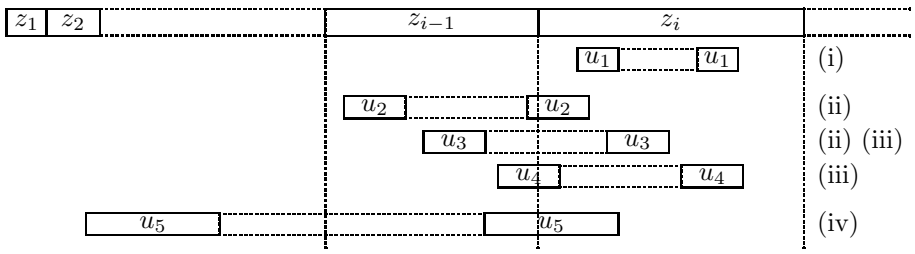


Figure 5.1: The only four possible locations of a repeat uvu involving phrase z_i of the factorisation of the string: (i) internal to z_i; (ii) the first occurrence of u is internal to z_{i-1}; (iii) the second occurrence of u is internal to z_i; (iv) the second occurrence of u is internal to $z_{i-1}z_i$.

We further distinguish four possible cases according to the position of the repeat uvu as shown in Figure 5.1.

(i) The two occurrences of u are contained in z_i.

(ii) First occurrence of u is contained in z_{i-1}, the second ends in z_i.

(iii) First occurrence of u starts in z_{i-1}, the second is contained in z_i.

(iv) First occurrence of u starts in $z_1 \cdots z_{i-2}$, the second is contained in $z_{i-1}z_i$.

Case (i) needs no action and other cases are handled by calls to Algorithm MaxExp as described in the algorithm below. There $\widetilde{x}$ denotes the reverse of string x. For any two strings z and w, and a positive rational number e, MaxExp(z, w, e) is the maximal exponent of repeats in zw whose occurrences start in z and end in w, and whose exponent is at least e; the output value is e if there is no such repeat.

MaxExpRep(y)

1 $(z_1, z_2, \ldots, z_k) \leftarrow$ f-factorisation of y
2 $\triangleright$ z_1 is the longest prefix of y in which no letter repeats
3 $e \leftarrow 1$
4 **for** $i \leftarrow 2$ **to** k **do**
5 $\quad e \leftarrow \max\{\text{MaxExp}(z_{i-1}, z_i, e), e\}$ (for case (ii))
6 $\quad e \leftarrow \max\{\text{MaxExp}(\widetilde{z_i}, \widetilde{z_{i-1}}, e), e\}$ (for case (iii))
7 $\quad$ **if** $i > 2$ **then**
8 $\quad\quad e \leftarrow \max\{\text{MaxExp}(\widetilde{z_{i-1}z_i}, z_1\widetilde{\cdots z_{i-2}}, e), e\}$ (for case (iv))
9 **return** e

Note that:

- As well as explanation in previous chapter, both $z_1\widetilde{\cdots z_{i-2}}$ and $\widetilde{z}_{i-2}\cdots\widetilde{z}_1$ refer to the reversal of $z_1\cdots z_{i-2}$, the first notation is more common to applied to the case (iv).
- Variable e can be initialised to be the repetitive threshold $\mathrm{RT}(a)$ of the alphabet of string y if the string is long enough. The maximal lengths of words containing no repeat of exponent at least $\mathrm{RT}(a)$ is 3 for $a = 2$, 38 for $a = 3$, 121 for $a = 4$, and $a + 1$ for $a \geq 5$ (see [25]).

Another technical remark: the instruction at line 6 can be tuned to deal only with type (iii) repeats of the form u_4vu_4 (see Figure 5.1), i.e. repeats for which the first occurrence of the border starts in z_{i-1} and ends in z_i, because line 5 finds those of the form u_3vu_3. But this has no influence on the asymptotic runtime.

Theorem 7. *For any input overlap-free string,* MaxExpRep *computes the maximal exponent of repeats occurring in the string.*

Proof. We consider a run of MaxExpRep(y). Let $e_1, e_2, \ldots, e_k$ be the successive values of the variable e, where e_i is its value just after the execution of lines 5–8 for index i. The initial value $e_1 = 1$ is the maximal exponent of repeats in z_1 as a consequence of its definition. We show that e_i is the maximal exponent of repeats occurring in $z_1z_2\cdots z_i$ if e_{i-1} is that of $z_1z_2\cdots z_{i-1}$, for $2 \leq i \leq k$.

To do so, since e_i is at least e_{i-1} (use of max at lines 5–8), all repeats occurring in $z_1 z_2 \cdots z_{i-1}$ are taken care of and we only have to consider repeats coming from the concatenation of $z_1 z_2 \cdots z_{i-1}$ and z_i, that is, repeats of the form uvu where the second occurrence of u ends in z_i. As discussed above and illustrated in Figure 5.1, only four cases are to be considered because the second occurrence of u cannot start in $z_1 z_2 \cdots z_{i-2}$ without contradicting the definition of z_{i-1}.

Line 5 deals with case (ii) by the definition of MaxExp. Similarly, line 6 is for case (iii), and line 8 for case (iv).

If a repeat occurs entirely in z_i, case (i), by the definition of z_i it occurs also in $z_1 z_2 \cdots z_{i-1}$, which is reported by e_{i-1}.

Therefore, all relevant repeats are considered in the computation of e_i, which is then the maximal exponent of repeats occurring in $z_1 z_2 \cdots z_i$. This implies that e_k, returned by the algorithm, is that of $z_1 z_2 \cdots z_k = y$ as stated. □

3 Repeats in a product

In this section we describe Algorithm MaxExp for computing the maximal exponent of repeats in a product of two strings zw that end in w, whose left border occurs in z, and whose exponent is at least e. MaxExp is called in the algorithm MaxExpRep of previous section.

To locate repeats under consideration, the algorithm examines positions j on w and computes for each the longest potential border of a repeat, a longest suffix u of $zw[0\,..\,j]$ occurring in z. The algorithm is built upon an algorithm that finds all of them using the Suffix Automaton of string z and described in [13, Section 6.6]. After u is located, some of its suffixes may lead to a repeat with a higher exponent, but the next lemmas show we can discard many of them.

Figure 5.2 illustrates the proof of the following lemma.

Lemma 5. *Let u' be a suffix of u. If they are both associated with the same state of the suffix automaton $\mathcal{S}(z)$ the maximal exponent of a $u'v'u'$ repeat is*

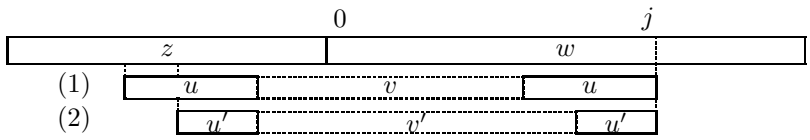


Figure 5.2: When u and its suffix u' end at the same rightmost position on z, repeat (1) has a larger exponent than repeat (2).

not greater than the maximal exponent of its associated uvu repeats.

Proof. The hypothesis implies that u and u' ends at the same positions in z, therefore they end at the same rightmost position (see Figure 5.2). Then, $u'v'u'$ and uvu have the same period but since u' is not longer than u, the exponent of $u'v'u'$ is not greater than that of uvu. □

Note that a suffix u' of u may have an internal occurrence in uvu, which would lead to a repeat having a larger exponent. For example, let $z =$ `abadba` and $w =$ `cdaba`. The repeat `abadbacdaba` with border `aba` has exponent 11/8 while the suffix `ba` of `aba` infers the repeat `bacdaba` of greater exponent 7/5.

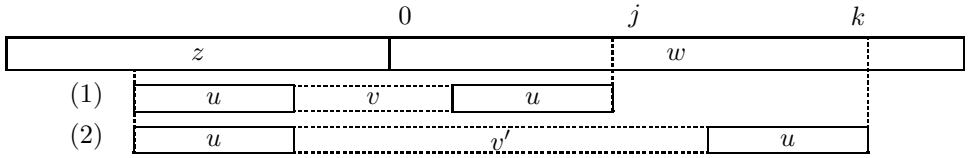


Figure 5.3: Repeat (1) ending at position j has a larger exponent than repeat (2) ending at position $k > j$.

Proof of the next lemma can be deduced from the remark in Figure 5.3.

Lemma 6. *If u occurs at end positions j and k on w with $k > j$, the repeat $uv'u$ ending at k cannot be a maximal-exponent repeat.*

Proof. To have a maximal exponent, the first occurrence of u in $uv'u$ should end at the right-most position on z. But then there is a repeat sharing the same first occurrence of u and with a closer second occurrence of u (see Figure 5.3). Therefore $1 + |u|/|uv| > 1 + |u|/|uv'|$, which proves the statement. □

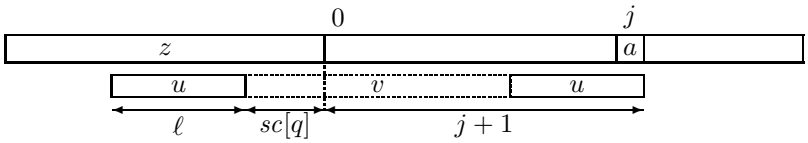


Figure 5.4: The maximal exponent of all repeats in question bordered by u, longest factor of z ending at j, is $(\ell + sc[q] + j + 1)/(sc[q] + j + 1)$.

The above properties are used by the algorithm MaxExp to avoid some exponent calculations as follows. Let uvu be a repeat ending at j on $zw[0\,..\,j]$ for which u is the longest string associated with state $q = \text{goto}(\text{initial}(\mathcal{S}), u)$. Then next occurrences of u and of any of its suffixes cannot produce repeats with exponent larger than that of uvu. State q is then marked to inform the next steps of the algorithm.

We utilise the suffix automaton of z, $\mathcal{S}(z)$, to locate borders of repeats. The structure contains the failure link F_z and the length function L_z both defined on the set of states. The link is defined as follows: let $p = \text{goto}(\text{initial}(\mathcal{S}(z)), x)$ for $x \in A^+$; then $F_z(p) = \text{goto}(\text{initial}(\mathcal{S}(z)), x')$, where x' is the longest suffix of x for which this latter state is not p. As for the length function, $L_z(p)$ is the maximal length of strings x for which $p = \text{goto}(\text{initial}(\mathcal{S}(z)), x)$.

We need another function, sc_z, defined on states of $\mathcal{S}(z)$ as follows: $sc_z(p)$ is the minimal length of paths from p to a terminal state; in other terms, if $p = \text{goto}(\text{initial}(\mathcal{S}(z)), x)$, then $sc_z(p) = |x'|$ where x' is the shortest string for which xx' is a suffix of z. With this precomputed extra element, computing an exponent is a mere division (see Figure 5.4).

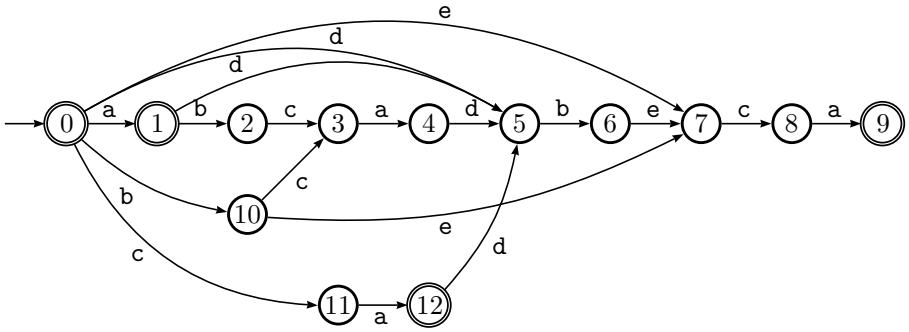


Figure 5.5: Suffix automaton of `abcadbeca`. Suffix links: $F[1] = 0$, $F[2] = 10$, $F[3] = 11$, $F[4] = 1$, $F[5] = 0$, $F[6] = 10$, $F[7] = 0$, $F[8] = 11$, $F[9] = 12$, $F[10] = 0$, $F[11] = 0$, $F[12] = 1$. Maximal incoming string lengths: $L[0] = 0$, $L[1] = 1$, $L[2] = 2$, $L[3] = 3$, $L[4] = 4$, $L[5] = 5$, $L[6] = 6$, $L[7] = 7$, $L[8] = 8$, $L[9] = 9$, $L[10] = 1$, $L[11] = 1$, $L[12] = 2$. Minimal extension lengths: $sc[0] = 0$, $sc[1] = 0$, $sc[2] = 7$, $sc[3] = 6$, $sc[4] = 5$, $sc[5] = 4$, $sc[6] = 3$, $sc[7] = 2$, $sc[8] = 1$, $sc[9] = 0$, $sc[10] = 3$, $sc[11] = 1$, $sc[12] = 0$.

MaxExp(z, w, e)

```
1  S ← Suffix Automaton of z
2  mark initial(S)
3  (q, ℓ) ← (F[last(S)], L[F[last(S)]])
4  for j ← 0 to min{⌊|z|/(e − 1) − 1⌋, |w| − 1} do
5      while goto(q, w[j]) = NIL and q ≠ initial(S) do
6          (q, ℓ) ← (F[q], L[F[q]])
7      if goto(q, w[j]) ≠ NIL then
8          (q, ℓ) ← (goto(q, w[j]), ℓ + 1)
9          (q′, ℓ′) ← (q, ℓ)
10         while q′ unmarked do
11             e ← max{e, (ℓ′ + sc[q′] + j + 1)/(sc[q′] + j + 1)}
12             if ℓ′ = L[q′] then
13                 mark q′
14             (q′, ℓ′) ← (F[q′], L[F[q′]])
15 return e
```

Figure 5.6 illustrates a computation done by the algorithm using the suffix automaton of Figure 5.5.

Note the potential overflow when computing $\lfloor |z|/(e-1)-1 \rfloor$ can easily be fixed in the algorithm implementation.

Theorem 8. *Algorithm* MaxExp*, applied to strings z and w and to the rational number e, produces the maximal exponent of repeats in zw that end in w, whose left border occurs in z and exponent is at least e.*

Proof. In the algorithm, position j on w stands for a potential ending position of a relevant repeat. First, we show that the algorithm does not need re-examine more values of j that those specified at line 4. The exponent of a repeat uvu is $|uvu|/|v|u$. Since we are looking for repeats satisfying exponent at least e that is $|uvu|/|vu| \geq e$, the longest possible such repeat has period $j+1$ and border z.

Then $(z+j+1)/(j+1) > e$ implies $j < z/(e-1) - 1$ (which is $+\infty$ if

j		0	1	2	3	4	5	6	7	8	9
$w[j]$		d	e	c	a	d	b	e	c	a	d
q	12	5	7	8	9	5	6	7	8	9	5
ℓ	2	3	1	2	3	3	4	5	6	7	3
exp		8/5	5/4	3/2	7/4	4/3	13/9	14/9	5/3	16/9	17/14
				5/4			10/9				

Figure 5.6: Computing exponents when searching zw for repeats uvu. The first occurrence of u is in z and the second ends in zw. The Suffix Automaton of $z =$ `abcadbeca` with function sc is shown in Figure 5.5. The search is done by parsing $w =$ `decadbecad` with the automaton. Exponents of repeats are given by the expression $(\ell + sc[q] + j + 1)/(sc[q] + j + 1)$. The last line is for exponents corresponding to suffixes of u. The maximal exponent of all repeats is 16/9.

$e = 1$). (That because $(z + j + 1)/(j + 1) > e$ implies $z/(j + 1) + 1 > e$, then $z/(e - 1) > j + 1$, hence $j < z/(e - 1) - 1$).)

Since j is a position on w, $j < w$, which completes the first statement.

Second, given a position j on w, we show that the algorithm examines all the possible concerned repeats having an exponent at least e and ending at j. The following property related to variables q, state of $\mathcal{S}$, and ℓ is known from [13, Section 6.6]: let u be the longest suffix of $zw[0\,..\,j]$ that is a factor of z, then $q = \text{goto}(\text{initial}(\mathcal{S}), u)$ and $\ell = |u|$. The property is also true just after execution of line 3 for z alone, due to the initialisation of the two variables.

Then string u is the border of a repeat ending in w and whose left border occurs in z. Lines 9 to 14 check the exponents associated with u and its suffixes. If q' is unmarked, the exponent is computed as explained before (see Figure 5.4). If the condition at line 11 is met, which means that u is the longest string satisfying $q' = \text{goto}(\text{initial}(\mathcal{S}), u)$, due to Lemma 6 the algorithm does not need to check the exponent associated with next occurrences of u, nor with the suffixes of u since they have been checked before. Due to Lemma 5, suffixes

of u ending at the same rightmost position on z do not have a larger exponent. Therefore the next suffix whose associated exponent has to be checked is the longest suffix leading to a different state of $\mathcal{S}$: it is $F(q')$ and the length of the suffix is $L(F(q'))$ by definition of F and L.

Finally note the initial state of $\mathcal{S}$ is marked because it corresponds to an empty string u, that is a repeat of exponent 1, which is not larger than the values of e.

This proves the algorithm runs through all possible relevant repeats, which ends the proof. □

4 Complexity analysis

In this section we analyse the running time and memory usage of our algorithms.

Proposition 3. *Applied to strings z and w and to the rational number e, Algorithm* MaxExp *requires $O(|z|)$ space in addition to inputs and runs in total time $O(|z| + \min\{\lfloor |z|/(e-1) - 1 \rfloor, |w| - 1\})$ on a fixed size alphabet. It performs less than $2|z| + \min\{\lfloor |z|/(e-1) - 1 \rfloor, |w| - 1\}$ exponent computations.*

Proof. The space is used mostly for storing the automaton, which is known to have no more $2|z|$ states and $3|z|$ edges (see [13]). It can be stored in linear space if edges are implemented by successor lists, which adds a multiplicative $\log a$ factor on transition time when a is the size of alphabet.

It is known from [13, Section 6.6] that the algorithm runs in linear time on a fixed alphabet, including the automaton construction with elements F, L and sc, if we exclude the time for executing lines 9 to 14.

So, let us count the number of times line 11 is executed. It is done once for each position j associated with an unmarked state. If it is done more than once for a given position, then the second value of q' comes from the failure link. A crucial observation is that condition at line 12 holds for such a state. Therefore, since $\mathcal{S}(z)$ has no more than $2|z|$ states, the total number of extra executions of line 11 is at most $2|z|$. Which gives the stated result. □

The proof of the linear running time of algorithm MaxExpRep additionally relies on a combinatorial property of strings. It is Dejean's statement [25] proved in [49, 24] that gives for each alphabet size a its repetitive threshold $\mathrm{RT}(a)$, i.e. the maximal exponent unavoidable in infinite strings over the alphabet. Thresholds are: $\mathrm{RT}(2) = 2$, $\mathrm{RT}(3) = 7/4$, $\mathrm{RT}(4) = 7/5$, and $\mathrm{RT}(a) = a/(a-1)$ for $a \geq 5$. Thus, if the string y is long enough, the maximal exponent of its factors is at least $\mathrm{RT}(a)$ where a is its alphabet size (see the note following algorithm MaxExpRep).

Theorem 9. *Applied to any overlap-free string of length n on a fixed-size alphabet, algorithm* MaxExpRep *runs in time $O(n)$ and requires $O(n)$ extra space.*

Proof. Computing the f-factorisation $(z_1, z_2, \ldots, z_k)$ of the input takes time and space $O(n)$ on a fixed-size alphabet using any suffix data structure. (It can even be done in time $O(n)$ on an integer alphabet, see [23].)

Next instructions execute in linear extra space from Proposition 3. Line 5 takes time $O(|z| + \min\{\lfloor |z_{i-1}|/(e-1) - 1 \rfloor, |z_i| - 1\})$, which is bounded by $O(|z_{i-1}| + |z_{i-1}|/(e-1) - 1)$, for $i = 2, \ldots, k$. For a long enough input, e is eventually at least $\mathrm{RT}(a)$ where a is the input alphabet. The time is then bounded by $O(|z_{i-1}| + |z_{i-1}|/(\mathrm{RT}(a) - 1) - 1)$, then $O(|z_{i-1}|)$ because $\mathrm{RT}(a) > 1$. The contribution of Line 5 to the total runtime is then $O(\Sigma_{i=2}^{k} |z_{i-1}|)$.

Similarly it is $O(\Sigma_{i=2}^{k} |z_i|)$ for Line 6 and $O(\Sigma_{i=2}^{k} |z_{i-1} z_i|)$ for Line 8. Thus the overall runtime is bounded by $O(\Sigma_{i=1}^{k} |z_i|)$, which is $O(n)$ as required. □

5 Enumeration of maximal-exponent repeats

In this section, we show there is a finite number (linear number in eventually) of maximal-exponent repeats in an overlap-free string on a fixed-size alphabet.

Note that on the alphabet $\{\mathtt{a}, \mathtt{a}_1, \ldots, \mathtt{a}_n\}$ the string $\mathtt{a}\mathtt{a}_1\mathtt{a}\mathtt{a}_2\mathtt{a} \ldots \mathtt{a}\mathtt{a}_{n-1}\mathtt{a}\mathtt{a}_n\mathtt{a}$ of length $2n+1$ has a quadratic number of maximal repeats. Indeed all occurrences of repeats of the form $\mathtt{a}w\mathtt{a}$ for a word w are non extensible. But only the n repeats of the form $\mathtt{a}c\mathtt{a}$ for a letter c have the maximal exponent $3/2$.

We start with a simple property of MERs, which does not prove their number is linear. However it is used below to tune the upper bound.

Lemma 7. *Consider two occurrences of MERs with the same border length b starting at respective i and j on y, $i < j$. Then, $j - i > b$.*

Proof. The two MERs having the same border length, since they have the same exponent, they have also the same period and the same length. Let b be their border length and p their period.

Assume ab absurdo $j-i \leq b$. The word $y[i\,..\,i+b-1] = y[i+p\,..\,i+p+b-1]$ is the border of the first MER. The assumption implies that $y[i+b] = y[i+p+b]$ because these letters belong to the border of the second MER. It means the first MER can be extended with the same period, a contradiction because it has the largest exponent. Therefore, $j - i > b$ as stated. □

If we count the occurrences of MERs by their border lengths after Lemma 7 we get an initial part of the harmonic series, quantity that is not linear with respect to the length y.

To refine the previous lemma and get a linear upper bound on the number of occurrences of MERs we introduce the notion of δ-MERs, for a positive real number δ: a MER uvu is a δ-MER if its border length $b = |u| = |uvu| -$ period(uvu) satisfies $3\delta \leq b < 4\delta$. Then any MER is a δ-MER for some $\delta \in \Delta$, where $\Delta = \{1/3, 2/3, 1, 4/3, (4/3)^2, (4/3)^3, \dots\}$. This is the technique used for example in [52, 15] to count runs in strings.

The proof of the next lemma is illustrated by Figure 5.7.

Lemma 8. *Let uvu and $u'v'u'$ be two δ-MERs starting at respective i and j on y, $i < j$. Then, $j - i \geq \delta$.*

Proof. Assume ab absurdo $j - i < \delta$ (see Figure 5.7).

Since both $|u| \leq 3\delta$ and $|u'| \leq 3\delta$, the two occurrences overlap. Let w be the overlap. It can be a suffix of u and a prefix of u' as in Figure 5.7, or w can be the shorter of u and u' when it occurs in the longer, see Figure 5.8. In both cases we have $|w| > 2\delta$.

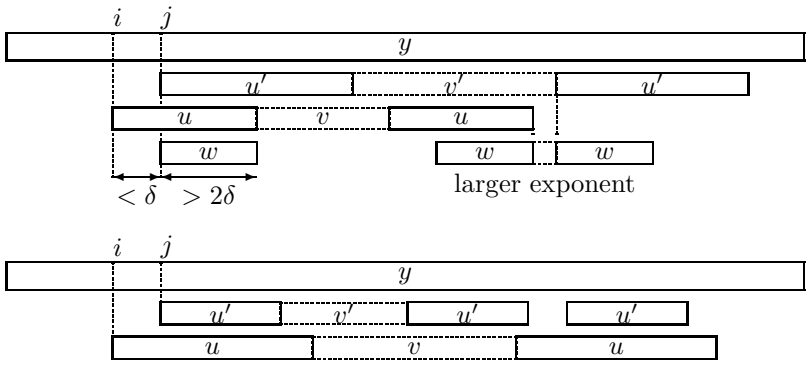


Figure 5.7: Top: two δ-MERs, uvu and $u'v'u'$, starting at close positions induce a repeat with a larger exponent, a contradiction. Bottom: the last two occurrences of u' are closer than the first two, leading to a larger exponent than $u'v'u'$, a contradiction. Indeed, the case is possible only if $|u'| \leq |u|/2$.

Let $p = |uv|$ be the period of uvu and $p' = |u'v'|$ be that of $u'v'u'$. Note that the exponent of the two repeats is $e = 1+|u|/p = 1+|u'|/p'$, which implies $p' - p = (|u'| - |u|)/(e-1)$.

Due to the periodicity of the two repeats, w occurs at both positions $j+p$ and $j+p'$. Assume for example that $j+p < j+p'$ (we cannot have $j+p = j+p'$). The factor $y[j+p\,..\,j+p+|w|-1]$ has exponent

$$1 + \frac{|w|}{p'-p} = 1 + \frac{|w|(e-1)}{(|u'|-|u|)}.$$

However since $|w| > 2\delta$ and $|u'| - |u| < 2\delta$, the exponent is larger than e, a contradiction with the definition of uvu and $u'v'u'$. Therefore $j - i \geq \delta$ as stated. □

A direct consequence of the previous lemma is the linearity of the number of MER occurrences.

Theorem 10. *There is a constant α for which the number of occurrences of maximal-exponent repeats in a string of length n is less than αn.*

Proof. Lemma 8 implies the number of δ-MER occurrences in y is no more than n/δ. Since values of δ in Δ cover all border lengths, the total number of

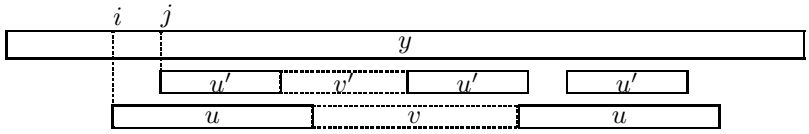


Figure 5.8: Second case of two δ-MERs, uvu and $u'v'u'$, starting at close positions: the last two occurrences of u' are closer than the first two, leading to a larger exponent than $u'v'u'$, a contradiction. Indeed, the case is possible only if $|u'| \leq |u|/2$.

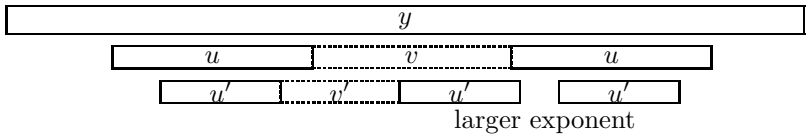


Figure 5.9: The left occurrence of u' from the MER $u'v'u'$ falls inside the left occurrence of u from the MER uvu. Then $|u'| \leq |u|/2$ as the contrary induces a repeat with a larger exponent, a contradiction.

occurrences of MERs is bounded by

$$\sum_{\delta\in\Delta}\frac{n}{\delta} = n\left(3+\frac{3}{2}+1+\frac{3}{4}+\left(\frac{3}{4}\right)^2+\dots\right) < 8.5\,n.$$

□

The next statement refines the upper bound given in the proof of Theorem 10.

Corollary 2. *There are less than* $3.11\,n$ *occurrences of maximal-exponent repeats in a string of length* n*.*

Proof. According to Lemma 7 there are less than

$$\sum_{b=1}^{b=11}\frac{n}{b+1} = 2.103211\,n$$

occurrences of MERs with border length at most 11.

We then apply Lemma 8 with values of $\delta\in\Gamma$ that allow to cover all remaining border lengths of MERs: $\Gamma = \{4, 4(4/3), 4(4/3)^2, \dots\}$, we get the upper bound

$$\sum_{\delta\in\Gamma}\frac{n}{\delta} = \frac{1}{4}\left(1+\frac{3}{4}+\left(\frac{3}{4}\right)^2+\dots\right)n = n$$

for the number of occurrences of MERs with border length at least 12.

Thus the global upper bound we obtain is $3.11\,n$. □

Note that the border length 11 (or 12) minimises the expression

$$\left(\sum_{b=1}^{b=k}\frac{n}{b+1}\right)+\frac{3}{k+1}\left(1+\frac{3}{4}+\left(\frac{3}{4}\right)^2+\dots\right)n = \left(\sum_{b=1}^{b=k}\frac{n}{b+1}\right)+\frac{12\,n}{k+1}$$

with respect to k, which means the technique is unlikely to produce a smaller bound. By contrast, experiments show that the number of occurrences of MERs is in fact smaller than n and not even close to n, at least for small values of n. Figure 5.10 displays the maximal number of MERs for overlap-free string lengths $n = 5, 6, \dots, 20$ and for alphabet sizes 2, 3 and 4. It also displays (second element of pairs) the associated maximal exponent. In the binary case

n	5	6	7	8	9	10	11	12
binary	$(2,2)$	$(3,2)$	$(4,2)$	$(5,2)$	5	6	6	8
ternary	$(2,1.5)$	$(3,1.5)$	$(4,2)$	$(5,2)$	$(5,2)$	$(6,1.5)$	$(6,2)$	$(8,2)$
$4-$ary	$(2,1.5)$	$(3,1.5)$	$(4,2)$	$(5,2)$	$(5,2)$	$(6,1.5)$	$(7,1.5)$	$(8,2)$

13	14	15	16	17	18	19	20
8	9	9	11	11	12	12	$(14,2)$
$(8,2)$	$(9,2)$	$(9,2)$	$(11,2)$	$(11,2)$	$(12,2)$	$(12,2)$	$(14,2)$
$(8,1.5)$	$(9,1.5)$	$(10,1.5)$	$(11,2)$	$(12,1.5)$	$(12,1.5)$	$(13,1.5)$	$(14,1.5)$

Figure 5.10: The maximal number of MERs and their maximal exponent for overlap-free string lengths $n = 5, 6, \ldots, 20$ and for alphabet sizes 2, 3 and 4.

we already know that it is 2 since squares are unavoidable in strings whose length is greater than 3.

6 Conclusion

The result of above implies that algorithm MaxExpRep can be modified to output all the MERs occurring in the input string in the same asymptotic time. Indeed, the only occurrences of MERs that are skipped by the algorithm when computing the maximal exponent are those occurring inside a phrase of the f-factorisation (Case (i) of Section 2). However storing the previous occurrences of MERs and listing them can be done in time proportional to their number, which does not affect the asymptotic running time of the algorithm and yields the next statement.

Corollary 3. *All the occurrences of maximal-exponent repeats of a string can be listed in linear time with respect to its length.*

The present work triggers the study of a uniform solution to compute both

repetitions and repeats. However, exponent 2 seems to reflect a transition phase in the combinatorics of these studied objects. For instance, the number of repetitions in a string can be of the order of $n \log n$, the number of runs is linear, while the number of repeats and of their maximal occurrences can be quadratic.

An interesting question is selecting repeats which occur only a linear number of times or slightly more. An attempt has been achieved in [40] where it is shown that the number of maximal repetitions of any exponent more than $1 + \epsilon$ is bounded by $\frac{1}{\epsilon} n \ln n$. See also the discussions at the end of [38] and of [14].

Another interesting problem is the calculation of the number of (distinct) MERs occurring in a string, as well as the lower bounds on these quantities.

6

Conclusion

In this chapter, we summarise results of the thesis and discuss the open problems as the future frameworks. To avoid confusion, the conclusion is presented by the sequence of the chapters of this thesis.

In Chapter 3, we presented a direct algorithm to compute all of the maximal periodicities, called runs, for a string drawn from an infinite alphabet. On a string of length n, the algorithm runs optimally in time $O(n \log n)$ although there is a linear number of runs. We also designed a time-optimal algorithm to compute all the local periods of a string, which additionally produces all its critical factorisations. To do so, we used the concept of a prefix table associated with a string for the design of string algorithms. We also studied a simple $O(n \log n)$ algorithm for computing the local periods of a string drawn from its Dictionary of Basic Factors (DBF).

Chapter 4 introduced gapped palindromes which are strings of the form $uv\widetilde{u}$ for some strings u, v with $|v| \geq 2$, and where $\widetilde{u}$ denotes the reversal of u. We also replicated the standard notion of string exponent and defined the anti-exponent of a gapped palindrome $uv\widetilde{u}$ as the quotient of $|uv\widetilde{u}|$ over $|uv|$. The techniques based on the suffix automaton and on the reversed Lempel-Ziv factorisation of an input string were applied in this chapter to design an algorithm to compute the maximal anti-exponent over all gapped palindromes

of the given string. Our algorithm runs in linear-time on a fixed-size alphabet in contrast to a naive cubic time solution.

One of the questions that remains is the calculation of the number of distinct maximal anti-exponent gapped palindromes occurring in a string. Another interesting open problem is the number of maximal anti-exponent unavoidable in gapped palindromes occurring in a string. Those two numbers are still unknown.

For Chapter 5, an algorithm to compute the maximal exponent of factors of an overlap-free string was created. The algorithm runs in linear-time on a fixed-size alphabet, while a naive solution of the question would run in cubic time. The solution for non overlap-free strings derives from algorithms to compute all maximal repetitions, also called runs, occurring in the string. We also showed there is a linear number of maximal-exponent repeats in an overlap-free string. The algorithm can locate all of them in linear time. However the number of (distinct) maximal-exponent repeats occurring in a string, as well as the lower bounds on these quantities, have not been calculated. Thus these are demonstrated as the open problems of the thesis.

Bibliography

[1] A. Apostolico and F. P. Preparata. Optimal off-line detection of repetitions in a string. *Theor. Comput. Sci.*, 22:297–315, 1983.

[2] G. Badkobeh and M. Crochemore. Computing maximal-exponent factors in an overlap-free string. *Journal of Computer and System Sciences*, 82(477–487), 2016.

[3] G. Badkobeh, M. Crochemore, and C. Toopsuwan. Computing the maximal-exponent repeats of an overlap-free string in linear time. In L. Calderon-Benavides, C. Gonzlez-Caro, E. Chvez, and N. Ziviani, editors, *Symposium on String Processing and Information Retrieval*, number 7608 in LNCS, pages 61–72. Springer, 2012.

[4] G. Badkobeh, M. Crochemore, and C. Toopsuwan. Maximal anti-exponent of gapped palindromes. In *Fourth International Conference on Digital Information and Communication Technology and its Applications DICTAP 2014, Bangkok, Thailand, May 6-8, 2014*, pages 205–210. IEEE, 2014.

[5] O. Berkman, C. S. Iliopoulos, and K. Park. The subtree max gap problem with application to parallel string covering. *Information and Computation*, 123(1):127–137, 1995.

[6] G. S. Brodal, R. B. Lyngsø, C. N. S. Pedersen, and J. Stoye. Finding maximal pairs with bounded gap. In M. Crochemore and M. Paterson, editors, *Combinatorial Pattern Matching*, volume 1645 of *Lecture Notes in Computer Science*, pages 134–149. Springer, 1999.

[7] G. S. Brodal and C. N. S. Pedersen. Finding maximal quasiperiodicities in strings. In R. Giancarlo and D. Sankoff, editors, *Combinatorial Pattern Matching*, number 1848 in LNCS, pages 397–411. Springer, 2000.

[8] S. Chairungsee and M. Crochemore. Efficient computing of longest previous reverse factors. In Y. Shoukourian, editor, Seventh International Conference on Computer Science and Information Technologies (CSIT 2009),

pages 27–30, Yerevan, 2009. The National Academy of Sciences of Armenia Publishers.

[9] M. Christou, M. Crochemore, C. S. Iliopoulos, M. Kubica, S. P. Pissis, J. Radoszewski, W. Rytter, B. Szreder, and T. Walen. Efficient seeds computation revisited. In R. Giancarlo and G. Manzini, editors, *Combinatorial Pattern Matching*, number 6661 in LNCS, pages 350–363, Berlin, 2011. Springer.

[10] M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Inf. Process. Lett.*, 12(5):244–250, 1981.

[11] M. Crochemore. Recherche linéaire d'un carré dans un mot. *C. R. Acad. Sc. Paris Sér. I Math.*, 296(18):781–784, 1983.

[12] M. Crochemore. Transducers and repetitions. *Theoret. Comput. Sci.*, 45(1):63–86, 1986.

[13] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007. 392 pages.

[14] M. Crochemore and L. Ilie. Maximal repetitions in strings. *Journal of Computer and System Sciences*, 74:796–807, 2008. DOI: 10.1016/j.jcss.2007.09.003.

[15] M. Crochemore, L. Ilie, and L. Tinta. The "runs" conjecture. *Theoretical Computer Science*, 412(27):2931–2941, 2011.

[16] M. Crochemore, C. S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, and T. Waleń. Extracting powers and periods in a string from its runs structure. In E. Chávez and S. Lonardi, editors, *SPIRE*, volume 6393 of *Lecture Notes in Computer Science*, pages 258–269. Springer, 2010.

[17] M. Crochemore, T. Kociumaka, W. Rytter, C. Toopsuwan, W. Tyczyński, and T. Waleń. *Algorithmics of repetitions, local periods and critical factorisation revisited*, pages 53–60. Czech Technical University, Prague, 2012.

[18] M. Crochemore and D. Perrin. Two-way string matching. *J. ACM*, 38(3):651–675, 1991.

[19] M. Crochemore and W. Rytter. Usefulness of the Karp-Miller-Rosenberg algorithm in parallel computations on strings and arrays. *Theoretical Computer Science*, 88(1):59 – 82, 1991.

[20] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, 1994.

[21] M. Crochemore and W. Rytter. *Squares, cubes, and timespace efficient strings searching*. Algorithmica 13, 1995.

[22] M. Crochemore and W. Rytter. *Jewels of Stringology*. World Scientific Publishing, Hong-Kong, 2002. 310 pages.

[23] M. Crochemore and G. Tischler. Computing longest previous non-overlapping factors. *Information Processing Letters*, 111:291–295, 2011.

[24] J. D. Currie and N. Rampersad. A proof of Dejean's conjecture. *Mathematics of Computation*, 80(274):1063–1070, 2011.

[25] F. Dejean. Sur un théorème de Thue. *Journal of Combinatorial Theory, Series A*, 13(1):90–99, 1972.

[26] J.-P. Duval, R. Kolpakov, G. Kucherov, T. Lecroq, and A. Lefebvre. Linear-time computation of local periods. *Theor. Comput. Sci.*, 326(1-3):229–240, 2004.

[27] K. Fan, S. J. Puglisi, W. F. Smyth, and A. Turpin. A new periodicity lemma. *SIAM J. Discrete Math.*, 20(3):656–668, 2006.

[28] A. S. Fraenkel and J. Simpson. How many squares can a string contain? *J. Comb. Theory, Ser. A*, 82(1):112–120, 1998.

[29] F. Franek, R. C. G. Fuller, J. Simpson, and W. F. Smyth. More results on overlapping squares. *J. Discrete Algorithms*, page to appear, 2012.

[30] F. Franek, R. J. Simpson, and W. F. Smyth. The maximum number of runs in a string, in: M. Miller, k. Park (eds.). *Proc. 14th Australasian Workshop on Combinatorial Algorithms*, pages 26–35, 2003.

[31] Z. Galil. Real-time algorithms for string-matching and palindrome recognition. In *Proceedings of the 8th Annual ACM Symposium on Theory of Computing, May 3-5, 1976, Hershey, Pennsylvania, USA*, pages 161–173, 1976.

[32] S. Grumbach and F. Tahi. Compression of DNA sequences. In J. A. Storer and M. Cohn, editors, *Proceedings of the IEEE Data Compression Conference DCC*, pages 340–350. IEEE Computer Society, 1993.

[33] D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology.* Cambridge University Press, Cambridge, 1997.

[34] C. S. Iliopoulos, D. Moore, and W. F. Smyth. A characterization of the squares in a Fibonacci string. *Theor. Comput. Sci.*, 172(1-2):281–291, 1997.

[35] C. S. Iliopoulos, D. W. G. Moore, and K. Park. Covering a string. *Algorithmica*, 16(3):288–297, 1996.

[36] D. E. Knuth, J. James H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[37] T. Kociumaka, J. Radoszewski, W. Rytter, and T. Walen. Efficient data structures for the factor periodicity problem. In *String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012, Cartagena de Indias, Colombia, October 21-25, 2012. Proceedings*, pages 284–294, 2012.

[38] R. Kolpakov and G. Kucherov. On maximal repetitions in words. *J. Discret. Algorithms*, 1(1):159–186, 2000.

[39] R. Kolpakov and G. Kucherov. Searching for gapped palindromes. In *Combinatorial Pattern Matching, 19th Annual Symposium, CPM 2008, Pisa, Italy, June 18-20, 2008, Proceedings*, pages 18–30, 2008.

[40] R. Kolpakov, G. Kucherov, and P. Ochem. On maximal repetitions of arbitrary exponent. *Information Processing Letters*, 110(7):252–256, 2010.

[41] R. M. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 596–604. IEEE Computer Society, 1999.

[42] E. Kopylova and W. F. Smyth. The three squares lemma revisited. *J. Discrete Algorithms*, 11:3–14, 2012.

[43] L. Lu, H. Jia, P. Dr"öge, and J. Li. The human genome-wide distribution of DNA palindromes. *Functional and Integrative Genomics*, 7(3):221–227, 2007.

[44] M. G. Main. Detecting leftmost maximal periodicities. *Discrete Applied Mathematics*, 25(1-2):145–153, Sept. 1989.

[45] M. G. Main and R. J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *J. Algorithms*, 5(3):422–432, 1984.

[46] G. Manacher. A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string. *J. ACM*, 22(3):346–351, July 1975.

[47] W. Matsubara, K. Kusano, A. Ishino, H. Bannai, and A. Shinohara. New lower bounds for the maximum number of runs in a string. In J. Holub and J. Žďárek, editors, *Proceedings of the Prague Stringology Conference 2008*, pages 140–145, Czech Technical University in Prague, Czech Republic, 2008.

[48] S. Puglisi, J. Simpson, and B. Smyth. How many runs can a string contain? *Theoret. Comput. Sci.*, 401(1-3):165–171, 2008.

[49] M. Rao. Last cases of Dejean's conjecture. *Theoretical Computer Science*, 412(27):3010–3018, 2011.

[50] S. Rozen, H. Skaletsky, J. Marszalek, P. Minx, H. Cordum, R. Waterston, R. Wilson, and D. Page. Abundant gene conversion between arms of

palindromes in human and ape Y chromosomes. *Nature*, 423(6942):873–876, 6 2003.

[51] W. Rytter. The number of runs in a string: Improved analysis of the linear upper bound, in: B. Durand, W. Thomas (eds.), Proc. of STACS'06. In *in:Lecture Notes in Comput. Sci*, volume 3884, pages 184–195, Berlin, Heidelberg, 2006. Springer.

[52] W. Rytter. The number of runs in a string. *Inf. Comput.*, 205(9):1459–1469, 2007.

[53] R. J. Simpson. Intersecting periodic words. *Theoret. Comput. Sci.*, 374:58–65, 2007.

[54] R. J. Simpson. Modified padovan words and the maximum number of runs in a word. *Australasian. J. Combinatorics*, 46:129–145, 2010.

[55] A. Thue. Über unendliche Zeichenreihen. *Norske Vid. Selsk. Skr. I Math-Nat. Kl.*, 7:1–22, 1906.

[56] I. H. W. Timothy C. Bell, John G. Cleary. *Text compression*. Prentice Hall, Englewood Cliffs, N.J., 1990.

[57] T. Tsunoda, M. Fukagawa, and T. Takagi. Time and memory efficient algorithm for extracting palindromic and repetitive subsequences in nucleic acid sequences. In *Proceedings of the 4th Pacific Symposium on Biocomputing PSB*, pages 202–213, 1999.

[58] P. E. Warburton, J. Giordano, F. Cheung, Y. Gelfand, and G. Benson. Inverted repeat structure of the human genome: The X-chromosome contains a preponderance of large, highly homologous inverted repeats that contain testes genes. *Genome Research*, 14:1861–1869, 2004.